Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Daniel Davis (B. Arch)

School of Architecture and Design
College of Design and Social context
RMIT University

February 2013

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Daniel Davis

February 2013

Abstract

In this thesis I consider the relationship between the design of software and the design of flexible parametric models.

There is growing evidence that parametric models employed in practice lack the flexibility to accommodate certain design changes. When a designer attempts to change a model's geometry (by modifying the model's underlying functions and parameters) they occasionally end up breaking the model. The designer is then left with a dilemma: spend time building a new model, or abandon the changes and revise the old model. Similar dilemmas exist in software engineering. Despite these shared concerns, Robert Woodbury (2010, 66) states that there is currently "little explicit connection" between the practice of software engineering and the practice of parametric modelling.

In this thesis I consider, using a reflective practice methodology, how software engineering may inform parametric modelling. Across three case studies I take aspects of the software engineering body of knowledge (language paradigms; structured programming; and interactive programming) and apply them to the design of parametric models for the Sagrada Família, the Dermoid pavilion, and the Responsive Acoustic Surface. In doing so I establish three new parametric modelling methods.

The contribution of this research is to show there are connections between the practice of software engineering and the practice of parametric modelling. These include the following:

- **Shared challenges:** Both practices involve unexpected changes occurring within the rigid logic of computation.
- **Shared research methods:** Research methods from software engineering apply to the study of parametric modelling.
- **Shared practices:** The software engineering body of knowledge seems to offer a proven pathway for improving the practice of parametric modelling.

These connections signal that software engineering is an underrepresented and important precedent for architects using parametric models; a finding that has implications for how parametric modelling is taught, how parametric models are integrated with practice, and for how researchers study and discuss parametric modelling.

Acknowledgements

Back in 2008 I found myself in a bar with my friend Kurt Rehder. We were talking about how my plans to spend the summer working in Melbourne were unravelling (I still had not found a job even though I was scheduled to travel there next week). I was explaining the start of the Global Financial Crisis and a multitude of other crises, but Kurt was not interested. He wanted to know who *I* most wanted to work with. I told him about Mark Burry and the Spatial Information Architecture Laboratory (SIAL). Kurt interrupted, "leave it with me." Although Kurt spoke with the authority of someone formally in the United States Navy, and although we had been at the bar long enough for his suggestion to sound plausible, I was also aware that Kurt knew no one in Melbourne and that I would need to invent a more feasible plan in the morning.

I awoke hazy and resumed my fruitless search for a summer job. A few days later a single-line email arrived, "Hi Daniel, We hear you are going to be in Melbourne, can you start on Monday – Mark."

Kurt never told me how he managed to get that email sent. I have often thought about it in my four years at SIAL. I have often thought about Kurt's insistence on aspiring towards something even when it feels unachievable. I have often thought about how small interactions can change the course of our lives. And here I would like to acknowledge some of the many small and large interactions with some of the people that have contributed to the course of my research. Save for my family, these are all people I would have never met had I not met Kurt in that bar back in 2008.

I am extremely grateful for Mark Burry and Jane Burry's supervision throughout my research. Together they have given me the latitude to explore while also giving me the critical grounding to ensure that I did not get too lost. In particular, Mark has been instrumental in facilitating many of the case studies my research examines. His humorous stories of parametric models failing in practice prompted much of this research. Jane has taught me a great deal about mathematics and made a number of important intellectual contributions to the project by quietly asking crucial questions at exactly the right time. Mark and Jane, together with John Frazer, were awarded the Australian Research Council Discovery Grant for *Challenging the Inflexibility of the Flexible Digital Model* that has funded my research.

A number of people at SIAL have helped shape my research, as collaborators on projects, as guides through institutional bureaucracy, and as sounding boards during chance meetings in corridors and formal presentations. In particular I would like to thank Alexander Peña de Leon, Chin Koi Khoo, Sascha Bohnenberger, Kamil Sharaidin, Nick Williams, Andrew Burrow, Flora Salim, Brad Marmion, Nicola Narayan, Susu Nousala, Juliette Peers, Margaret Woods, Dominik Holzer, Tim Schork, and Inger Mewburn. I am also thankful to the people who have visited SIAL and critiqued my thesis as part of RMIT's Graduate Research Conference: Jeff Malpas, Terry Cutler, Michael Ostwald, Anthony Burke, Jill Franz, Jules Moloney, Ken Friedman, Margot Brereton, and Tom Daniell.

My thesis was examined by Sean Hanna and an individual who chose to remain anonymous. Their comments have been crucial in adding the final polish and I am appreciative of the time they both invested in closely reading this thesis.

Many of the projects in this thesis have taken place at Center for Information Technology and Architecture (CITA) in the Royal Danish Academy of Fine Arts. Mette Thomsen has generously welcomed me into CITA where I have spent many enjoyable days working with Martin Tamke, Brady Peters, Phil Ayres, Anders Deleuran, Aron Fidjeland, Stig Nielsen, Morten Winter, Tore Banke, Jacob Riiber, and John Klein.

My friends have been extremely supportive. Together we have despaired and joked, enjoyed meals and been on adventures. A number of them have also read and edited my thesis. I would especially like to thank Kira Randolph, Jordana Aamalia, Luke Feast, Byron Kinnaird, Rhys Williams, Ross Smith, Agnes So, and Richard Maddock. And, of course, my family: Lloyd Davis, Frances McCaffrey, and Kelsey Davis.

Finally, I would like to dedicate this thesis to Kurt Rehder (1969-2010).

Contents

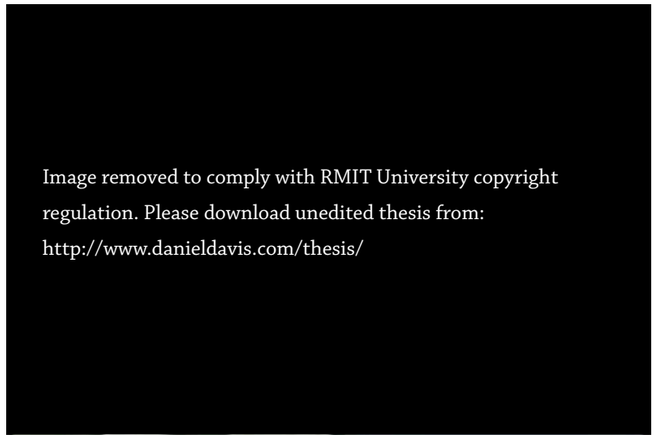


Figure 1: In a frame from the Lincoln Laboratory (1964) documentary, Timothy Johnson draws a line with Sketchpad's light-pen.

1 Introduction

Timothy Johnson (fig. 1; Lincoln Laboratory 1964) drags a pen across a flickering screen and, for a brief instance, it looks as though he is drawing. For most people in 1964, this demonstration of Ivan Sutherland's (1963) Sketchpad will be the first time they have seen someone interact with a computer. Johnson's actions initially resemble a designer at a drawing board, the ink replaced with light, the impression of the computer's similarity to the drawing board only broken when Johnson creates a misshapen trapezium (fig. 2) and defiantly states "I shouldn't be required to draw the exact shape to begin with" (Lincoln Laboratory 1964, 9:00). He then obscures the screen by reaching for a button to his left (fig. 3). When Johnson sits back down he reveals, with a hint of magic, that the trapezium has transformed into a rectangle (fig. 4). The sleight of hand underlying this trick is that when Johnson appears to be drawing he is actually communicating a set of geometric relationships to the computer. These relationships form a specific type of computer program, known today as a parametric model. Changes to the model or to its input variables propagate through explicit functions in the computer to change the model's output automatically, which allows trapeziums to transform into squares, and allows designers to begin drawing before they know the exact shape.

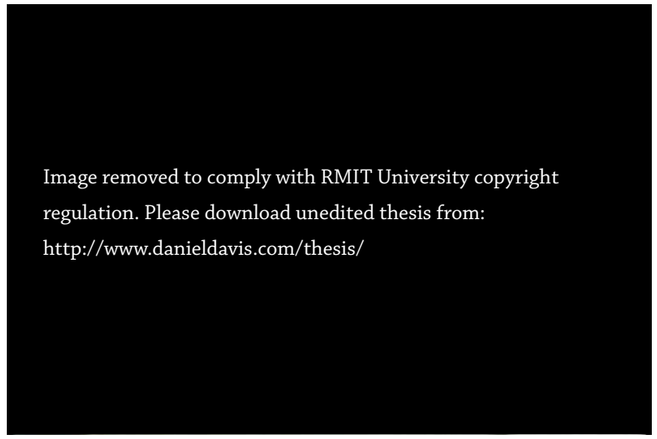


Figure 2: A trapezium drawn in Sketchpad.

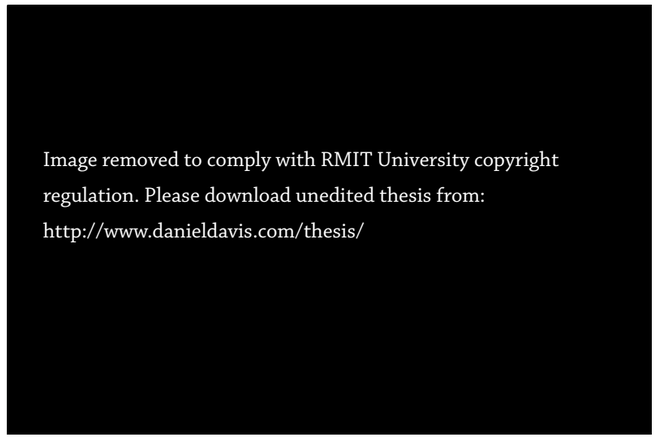


Figure 3: Timothy Johnson reaches for a button (off camera) to invoke Sketchpad's parametric engine.

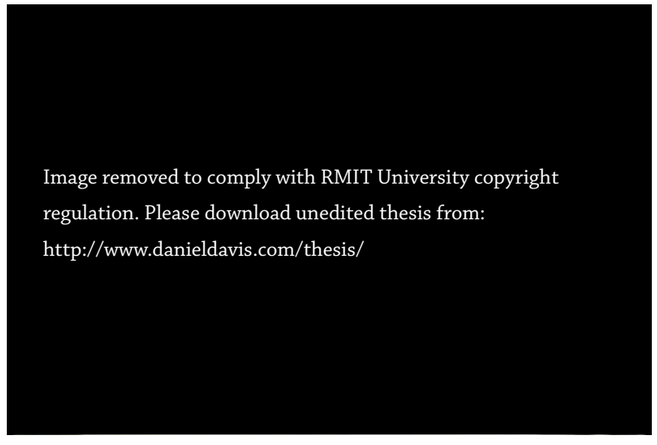


Figure 4: The model's explicit functions create orthogonal angles between the trapezium's sides, transforming it into a rectangle.

When Sutherland created Sketchpad, it was optimistic to think designers would gain flexibility by transforming a job a draftsperson did with a sheet of paper into a job a software engineer (Timothy Johnson) did with a computer large enough to require its own roof (the TX-2). Yet, fifty years later, architects sit at computers and increasingly do exactly what Johnson was doing all those years ago. It is a difficult job. I will reveal in this thesis that while parametric models offer the flexibility to accommodate change, there is a tendency for changes to break parametric models unexpectedly. These problems resemble problems software engineers encounter when programming; problems that have led designers as far back as 1993 to declare that parametric modelling is "more similar to programming than to conventional design" (Weisberg 2008, 16.12). Yet despite the shared concerns with accommodating changes, there is currently, according to Robert Woodbury (2010, 66), "little explicit connection" between the practice of parametric modelling and the practice of software engineering. Johnson's sleight of hand persists. In this thesis I seek to connect these two practices by considering whether parametric models can be made more flexible through the application of knowledge from software engineering.

1.1 Problems with Flexibility

Design is a journey traced by ever changing representations. "The designer sets off," Nigel Cross (2006, 8) argues, "to explore, to discover something new, rather than to return with yet another example of the already familiar." Many others share Cross's view that design is not simply a leap into a premeditated solution but rather a messy journey necessitated by uncertainty and characterised by iteration (Schön 1983; Lawson 2005; Brown 2009). Key facilitators of this process are external representations, which serve as points of reflection along the way (Schön 1983). Cross (2011, 12) contends that these representations are necessary since "designing, it seems, is difficult to conduct by purely internal mental processes." Yet representations take time to produce, and they take time to modify. Thus, as change inevitably occurs whist designing, the designer necessarily spends time changing or creating new representations.

From the very beginning, the digitisation of architecture has concerned itself with facilitating changes to design representations. When Sutherland created Sketchpad, he spent considerable time considering how a "change of a [model's] critical part will automatically result in appropriate changes to related parts" (Sutherland 1963, 22). In today's lexicon this could be described as parametric, meaning the geometry (the related part) is an explicit function of a set of parameters (the critical part). As Johnson demonstrated, a designer using Sketchpad could change their mind about the relationship between objects (the critical part) and Sketchpad would automatically adapt the objects (the related parts) to satisfy this relationship. For example, a designer might decide that two sides of trapezium should be orthogonal (as Johnson does in figure 2). The designer then modifies the parameter controlling line relationships, and this change filters through the explicit functions in Sketchpad's parametric engine to trigger the automatic remodelling of the lines so that they meet orthogonally; the trapezium now a square. By making geometric models seemingly easy to change, Sketchpad and the introduction of computer-aided design promised to reduce the hours designers spent manually changing or creating new representations.

It is now almost fifty years after Sketchpad and computers have replaced the drawing boards they once imitated. Many new ways of generating parametric models have been developed: from history-based modellers (CATIA, SolidWorks, and Pro/Engineer), to visual scripts (Grasshopper, GenerativeComponents, and Houdini) and textual programming environments (the scripting interfaces to most CAD programs). The commonality of all these parametric modelling environments is the ability for designers to modify parameters and relationships that trigger the transformation of related model parts. This is now a popular way to create and modify digital models. Fifty years since Sketchpad, Robert Aish and Robert Woodbury (2005, 151) say the hope still remains that parametric modelling will "reduce the time and effort required for change and reuse."

While it is alluring to think of a design representation flexible enough to reduce time spent remodelling, the reality – a reality commonly not addressed – is that parametric models are often quite brittle. Frequently I find my models have grown so tangled they can no longer accommodate even the most trivial change. Often I just start again. I am not alone in experiencing this: I see my students work themselves into the same situation, I listen to colleagues complain about their models breaking, and I read firsthand accounts from architects detailing the same problem.

> When the topology of a project changes the [parametric] model generally needs to be remade.
> *David Gerber 2007, 205*

> A designer might say I want to move and twist this wall, but you did not foresee that move and there is no parameter to accommodate the change. It then unravels your [parametric model]. Many times you will have to start all over again.
> *Rick Smith 2007, 2*

> To edit the relational graph or remodel completely is also commonplace.
> *Jane Burry 2007, 622*

> Changes required by the design team were of such a disruptive nature that the parametric model schema could not cope with them. [They had to rebuild part of the model.]
> *Dominik Holzer, Richard Hough, and Mark Burry 2007, 639*

> [Parametric modelling] may require additional effort, may increase complexity of local design decisions and increases the number of items to which attention must be paid in task completion.
> *Robert Aish and Robert Woodbury 2005, 151*

> [If a critical change is made] there is no solution other than to completely disassemble the model and restart at the critical decision.
> *Mark Burry 1996, 78*

These authors collectively demonstrate that parametric models used in practice have a propensity for inflexibility, which sometimes leads the model to break (these quotes and their implications are examined further in chapter 2). Often the only way forward is rebuilding the brittle part of the model. In the best case, this causes an unexpected delay to the project. In the worst case, the designer is dissuaded from making the change and ends up with a design that was not so much created in a parametric model as it was created for the limitations of a parametric model. Despite fifty years of refinement and the increased adoption of parametric modelling, inflexible models still cause delays in architectural practice, hinder collaboration, and discourage designers from making changes to their models.

1.2 The Flexibility of Code

In *The Design of Design* the Turing award-winning software engineer, Frederick Brooks (2010), frequently cites Schön and Cross as he positions programming as a design discipline. Brooks argues that programming, like other forms of design, is not simply a leap into a premeditated solution but rather a messy journey necessitated by uncertainty and characterised by iteration. During this process software engineers represent ideas with computer code, which primarily contains relationships expressed in a logical syntax. As these networks of relationships develop during the design process, they can become brittle and unexpectedly break. These moments of inflexibility echo moments of inflexibility exhibited by some parametric models. For a period in the 1960s, scientists feared these brittle breakages would be insurmountable and the limits of computation would not be computer speed but rather the cognition of the programmers creating and maintaining software (Naur and Randell 1968, chap. 7.1).

Software engineers still struggle with inflexibility. While it is difficult to get precise data, some industry surveys have suggested only 14% to 42% of software projects were successful in 2012 (The Standish Group 2012; The Standish Group 2009; Eveleens and Verhoef 2010). In a discipline where the "costs of materials and fabrication are nil," Young and Faulk (2010, 439) argue that the primary determinants of a programmer's success become "the essential challenges of complexity and the cost of design." Accordingly, the *Software Engineering Body of Knowledge Version 1.0* (Hilburn et al. 1999)

– which categorises the expected knowledge of a programmer – is filled with management strategies, algorithms and data structures, problem decomposition tactics, programming paradigms, and coding interfaces to reduce both the complexity and the cost of design. This is the cumulative wisdom from years of struggles with inflexibility and these are the issues programmers talk about.

In contrast, architects are told by Mark Gage (2011, 1), the assistant dean at the Yale School of Architecture, to "use computation, but stop fucking talking about it." Gage (2011, 1) goes on to parody the way architects talk, justifying computational projects as coming from "secret code found in the lost book of the Bible handed to [them by their] Merovingian great grandmother" (2011, 1) or deriving from "a semester producing the most intricate parametric network ever seen by man" (2011, 1). While there is obviously an element of truth to Gage's polemic caricature, it does not necessarily support his conclusion that architects should stop talking about computation. Robert Woodbury (2010, 66) has pointed out, "parametric modelers do have common cause with professional programmers" but "there is little explicit connection between them." From this point of view, the problem is not so much that architects are talking about computation, but rather that architects are typically talking about computation in fairly extraneous terms compared to the exchanges characteristic of software engineers.

In this thesis I explore whether the debates surrounding the design of software are applicable to the design of flexible parametric models. I position change as an essential, desirable, and unavoidable aspect of both software design and parametric design – a quality both disciplines stiffly embrace in their practice. The relationship between the two disciplines is traversed in my research through three case studies, which take methods inspired from software engineering and apply them to the creation of parametric models. These three case studies map a new territory for architects; territory that concerns the way parametric models themselves are structured and considers what architects can learn from software engineers to improve the flexibility of their parametric models in the face of change.

1.3 Aim

The aim of this research is to explore whether the design of software can inform the design of flexible parametric models.

In addressing this aim, my thesis occupies what Woodbury (2010, 66) has identified as the "common ground" between programming and parametric modelling, a space where currently "little explicit connection" exists between either side. In this thesis I consider how practices from software engineering may connect with practices of parametric modelling, which I do by applying software engineering concepts to the creation of parametric models. In chapter 3 I discuss which software engineering concepts may also be applicable to the practice of parametric modelling. I then select three of these concepts and apply them respectively to three parametric architecture projects in chapters 5, 6, & 7. The concluding chapters (chap. 8 & 9) bring these case studies together to consider how the design of software may inform the design of flexible parametric models and to consider the consequences of this relationship for architecture generally.

A limit of my aim is that it explores just one type of flexible digital modelling: parametric modelling. Not all architects use parametric models and for those that do, parametric modelling is but one technique in an array of modelling methods available to them. Flexibility may also be achieved through other methods like Building Information Modelling (BIM). The many advocates of BIM contend that BIM reduces rework (which creates flexibility) by creating a centralised repository of data that all representations draw upon; change the data once and theoretically everything updates (Eastman et al. 2011, 15-26). My research focuses on parametric modelling due to my experience with it and due to the opportunities for improvement this presents. However focusing solely on parametric modelling is not intended to be antagonistic to the other modelling methods. On the contrary, since architects often integrate modelling methods together, improvements to the practice of parametric modelling could manifest themselves as improved parametric features for other flexible representations like BIM.

A Note on Language

Within this aim, and throughout this thesis, I frequently use the terms *parametric model* and *software engineering*. Both are contentious. To help clarify my intended meaning I will briefly pause here and explain why I have included these terms.

Architects assign a range of meanings to the phrase *parametric model*. A number of definitions have been advanced that range from style-based classifications, to software zealotism, to arguing that all design is inherently parametric. Neil Leach claims that the disagreement is so fierce "few people in the West working at the forefront of computation use the term *parametric*" (Leach and Schumacher 2012). I however have chosen to use *parametric* in this thesis because *parametric* has a very precise historic meaning. The range of contemporary definitions are an illustration of how the modern conception of parametric modelling has shifted. I explore these shifts further in chapter 2. In the same chapter I explain the definition I use in this thesis: a parametric model is set of equations that express a geometric model as explicit functions of a number of parameters.

"The phrase *software engineering* was deliberately chosen as being provocative" write the authors who coined the term at the 1968 meeting of the NATO science committee (Naur and Randell 1968, 13). The original intention was to ground the practice of manufacturing software in a theoretical foundation similar to other engineering disciplines (Naur and Randell 1968, 13). Many have since argued that engineering is an inappropriate discipline to base the identity of programming upon. This has led Tom DeMarco (2009, 95) to declare "software engineering is an idea whose time has come and gone." Others have said the manufacture of software is more like a design discipline (Brooks 2010), or a craft (Seibel 2009), or an art (Knuth 1968). It lies outside the scope of my research to resolve this forty-year old debate. In this thesis I use the phrase *software engineering* not because it is an apt analogy for what programmers do but rather because *software engineering* is a term still a widely used to denote the body of knowledge concerning the creation of software.

1.4 Methodology

Previous research indicates there are methodological challenges in developing a convincing understanding of flexible parametric model design. When software engineers have sought similar understandings of software design, the researchers have shown a tendency to seek elegant, repeatable, statistical studies – perhaps owing to the mathematical and scientific origins of computer science (Menzies and Shull 2010, 3). Achieving this elegance, repeatability, and statistical confidence often requires the simplification of complicated circumstances. Turing award-winner Edsger Dijkstra (1970, 1) has stated that these simplifications inevitably lead computer scientists to conclude with the assumption: "when faced with a program a thousand times as large, you compose it in the same way." For certain problems this extrapolation works, but on problems concerning software flexibility and maintainability, Dijkstra (1970, 7) argues idealised experiments fail to capture the paramount issues of "complexity, of magnitude and avoiding its bastard chaos." In other words, simplifying, controlling, and isolating the issues of practice with a positivist or post-positivist perspective may generate convincing empirical evidence for software engineering researchers, but there is reason to suspect the simplifications will also abstract away the crucial parts of what needs to be observed to produce convincing evidence for practitioners.

With complicated interrelationships and the pressures of practice likely to be important components of this research, a primary consideration is how to avoid obscuring the nuances of practice whilst observing it. One method is to conduct the investigation from within practice, a method Schön (1983) describes as *reflection in action* and *reflection on action*. This method has a constructivist worldview where, according to Creswell and Clark (2007, 24), multiple observations taken from multiple perspectives build inductively towards "patterns, theories, and generalizations." While this may be closer to social science than the *hard science* origins of software engineering, Andrew Ko (2010, 60) argues such an approach is "useful in any setting where you don't know the entire universe of possible answers to a question. And in software engineering, when is that *not* the case?" The challenges of understanding practice therefore becomes one of generalising results that are not necessarily representative because they are based on observations of projects that cannot be simplified, controlled, and isolated. To help

mitigate these challenges my research draws upon multiple research instruments to make the observations, and multiple case studies to triangulate the results.

1. **Multiple case studies:** By employing multiple case studies, the anomalies of one can be balanced by the rest. Robert Stake (2005, 446) calls this a "collective case study" where multiple projects "are chosen because it is believed that understanding them will lead to better understanding, and perhaps better theorising, about a still larger collection of cases." Chapter 4 discusses in greater detail the criteria for selecting the three case studies.
2. **Multiple research instruments:** A research instrument, as defined by David Evan and Paul Gruba (2002, 85), is any technique a "scientist might use to carry out their 'own work'." Typical examples include interviews, observations, and surveys. Unfortunately there is no research instrument to measure parametric flexibility. Chapter 4 investigates how various qualitative and quantitative research instruments, many borrowed from software engineering, can aid observations of parametric flexibility. These are combined in various ways within the case studies to present a fuller picture of the various projects.

Whilst this triangulation of observations through a mix of research instruments is not as precise as a controlled experiment, it does aid in observing the influence of actions undertaken in the midst of large, messy, and complicated practice based projects – the situations where the flexibility of parametric models is critical.

I should also note (prior to discussing the thesis structure) that the chapter sequence in this thesis does not trace how I conducted the research. As a work of reflective practice I gathered the evidence of my research in a process resembling Kemmis and McTaggart's (1982) cycle of "planning, acting, observing and reflecting" on actions in practice. In this sense my thesis represents a final extended reflection on my prior cycles of research. I use three of these cycles as case studies but there are also many incomplete and tangential cycles that are left unstated. As such, my thesis structure does not mirror my research process, and instead it follows a logic intended to contextualise the case studies in order to reflect upon the relationship between software engineering and parametric model design.

1.5 Thesis Structure

This thesis is divided into nine chapters: the current introduction, three background chapters, three case study chapters, and two concluding chapters.

In the following chapter (chap. 2) I expand upon the challenges associated with parametric modelling that I have outlined in this introduction. I first examine the various definitions of parametric modelling and consider how these frame an understanding of what a parametric model is. I go on to reveal the numerous challenges architects have faced when using parametric models in practice. Aggregated together, these accounts reveal an array of problems that tend to be overlooked in many of the discussions around parametric modelling.

In chapter 3 I contrast the challenges of parametric modelling to the challenges associated with software engineering. I introduce the body of knowledge associated with software engineering and hypothesise about which knowledge areas may also help the practice of parametric modelling.

In chapter 4 I discuss a research method for applying aspects of the software engineering body of knowledge to the creation of various parametric models. I outline criteria for selecting the case studies and I discuss how a variety of quantitative and qualitative metrics can be used to observe parametric flexibility.

Each of the subsequent three chapters is a case study that takes an area of knowledge identified in chapter 3 and observes impact on parametric modelling with techniques from chapter 4.

In chapter 5 I explore the differences between creating a parametric model with a logic programming paradigm compared to creating a model with a more conventional dataflow paradigm. The logic programming paradigm enables the reversal of the parametric process by turning static geometry into a parametric model. However, outside this niche application, logic programming proves to be a difficult modelling interface.

In chapter 6 I consider how the principles of structured programming apply to the organisation of parametric models. Splitting models into hierarchies of modules appears to increase the legibility of the models and improve model reuse. Perhaps more importantly, the structure seemed to allow ordinarily pivotal decisions to be made much later in the design process – in some cases, moments prior to construction.

In chapter 7 I draw upon innovations in software engineering Integrated Development Environments (IDEs) to create an interactive programming interface for architects. The interface enables designers to modify their code and immediately see the geometry of the model change. This case study positions the scripting environment itself as a important site of innovation, a site where many programmers have already provided numerous useful innovations.

These three chapters feed into the discussion (chap. 8) and conclusion (chap. 9). I argue there is a close relationship between software engineering and parametric modelling. This relationship has implications for how parametric modelling is taught, for how parametric modelling is integrated in practice, and for how we discuss parametric modelling.

2 The Challenges of Parametric Modelling

Neil Leach observes that "many people have misgivings about the term *parametric*" (Leach and Schumacher 2012). Whilst writing this thesis I contemplated avoiding any controversy by replacing *parametric* with a less disputed synonym: associative geometry, scripting, flexible modelling, algorithmic design. I would not be the first author to shy away from a term that others, like Patrik Schumacher (2010), have declared "war" over.[1] The battles and misgivings about the term *parametric* are relatively recent. They signify, if nothing else, the growing importance of parametric modelling within the discourse of architecture – now important enough for well known architects to go to war over. These challenges in defining *parametric* also help explain some of the challenges of using parametric models. For this reason, I now dedicate a chapter to unpacking the term *parametric*, which I will use throughout this thesis.

Owen Hatherley (2010) argues that the debates surrounding parametric modelling stem from a shift in definition. The term *parametric* was co-opted, says Hatherley (2010), from its provenance in the "digital underground" by "arrivistes" who have jostled to claim the term as their own whilst parametric design ascended towards "mainstream acceptance." Hatherley cites Schumacher as an example of an *arriviste*, owing to Schumacher's (2008) infamous claim that parametric design is a "contemporary architectural style that has achieved pervasive hegemony within the contemporary architectural avant-garde." Hatherley (2010) goes on to quote my previous articles – where I have argued that parametric design is not defined by an architectural style (Davis 2010) – as "perhaps the nearest proof that there really is an avant-garde [of parametric design] although perhaps Schumacher has little to do with it." I will make a similar argument in this chapter by showing how the definition of *parametric* has

1 Neil Leach claims that, as a result of the misgivings around the term *parametric*, "few people in the West working at the forefront of computation use the term *parametric* or *parametricism*, although it is still popular in China for some reason" (Leach and Schumacher 2012).

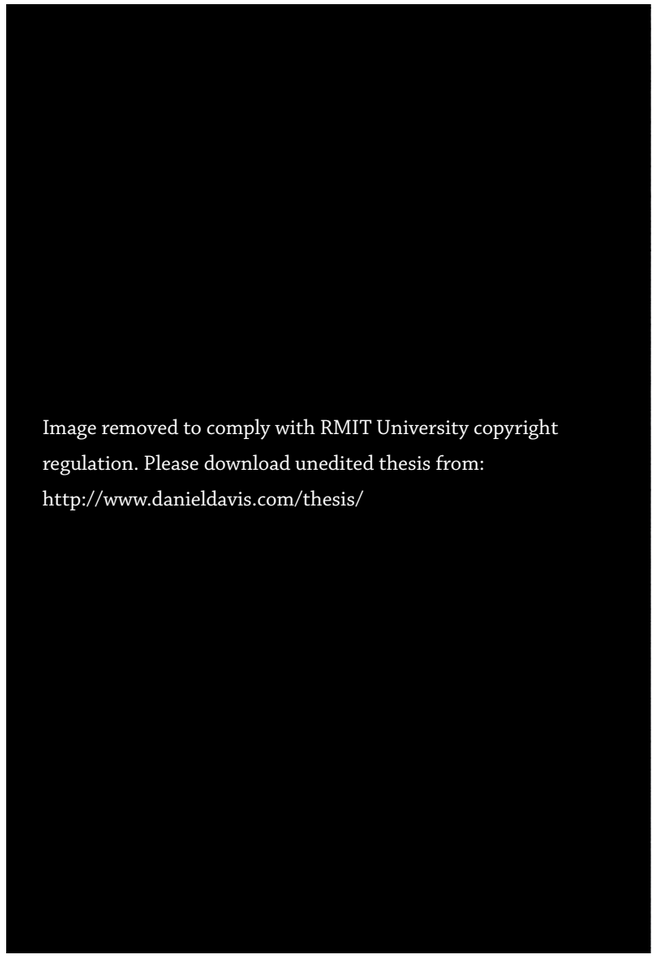
shifted and, in doing so, obscured many of the challenges associated with parametric modelling.

I begin this chapter by exploring the various definitions of parametric modelling. I argue that there is a propensity to define parametric modelling in terms of the model's outputs even though the defining feature of a parametric model is the need to construct and maintain relationships associated with the model. I go on to explore why architects are attracted to this seemingly unintuitive design process and I investigate the difficulties architects encounter with using parametric models. I argue that these difficulties are not necessarily obvious if parametric modelling is only defined and understood in terms of the model's outputs. Accordingly, I spend most of this chapter discussing the challenges of defining and using parametric models, both as a way to position my research and as a way to highlight under-represented parts of the discourse that are important in understanding why architects sometimes find parametric modelling challenging.

2.1 What is Parametric Modelling?

"What is parametric modelling?" is the title that heads the second chapter in Robert Woodbury's (2010) book *Elements of Parametric Design*. Woodbury dedicates twelve pages to the question, but instead of directly answering the question he spends most of these pages explaining the workings of forward-propagating parametric models. Woodbury's most forthright answer appears on the chapter's first page (fig. 5): "parametric modelling introduces fundamental change: 'marks', that is, *parts of the design*, relate and change together in a coordinated way" (Woodbury 2010, 11). But Woodbury never pauses to explain how relating marks together differs from the relationships found in a plethora of alternative modelling methods, notably BIM. This is not to chastise Woodbury, for *Elements of Parametric Design* is one of the seminal books on parametric modelling, but this is to highlight the difficulty even experts have in articulately answering basic questions like *what is parametric modelling?*

Figure 5: The eleventh page from Robert Woodbury's (2010) *Elements of Parametric Design*. Woodbury asks "what is parametric modelling?" but never quite gives the answer.



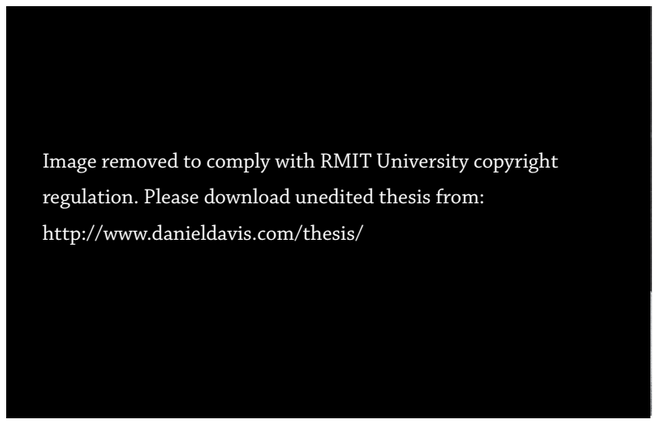


Figure 6: Stadium designs by Luigi Moretti from the 1960 *Parametric Architecture* exhibition at the Twelfth Milan Triennial. Each stadium derives from a parametric model consisting of nineteen parameters. Top: The plans for stadium version *M* and *N* showing the "equi-desirability" curves (Converso and Bonatti 2006, 243) Bottom: A model of stadium *N*.

Defining what parametric modelling is and what makes it unique, is an important first step towards identifying the idiosyncratic challenges parametric models present. In the following pages I traverse a range of definitions that various architects have put forward: from the historic definition of parametric, through to the claims that all design is parametric, and that either change, tooling, or parametricism defines parametric. In doing so I make the case that many contemporary definitions tend to privilege what parametric models do (in terms of model behaviour or stylistic outcomes) but that it is how parametric models come to be (through the construction and maintenance of relationships) that distinguishes parametric modelling from other forms of architectural representation.

A Historic Definition

The term *parametric* originates in mathematics but there is debate as to when designers initially began using the word. David Gerber (2007, 73), in his doctoral thesis *Parametric Practice*, credits Maurice Ruiter for first using the term in a paper from 1988 entitled *Parametric Design*.[2] 1988 was also the year Parametric Technology Corporation (founded by mathematician Samuel Geisberg in 1985) released the first commercially successful parametric modelling software, Pro/ENGINEER (Weisberg 2008, 16.5). But Robert Stiles (2006) argues that the real provenance of *parametric* was a few decades earlier, in the 1940s' writings of architect Luigi Moretti (Bucci and Mulazzani 2000, 21). Moretti (1971, 207) wrote extensively about "parametric architecture," which he defines as the study of architecture systems with the goal of "defining the relationships between the dimensions dependent upon the various parameters." Moretti uses the design of a stadium as an example, explaining how the stadium's form can derive from nineteen parameters concerning things like viewing angles and the economic cost of concrete (Moretti 1971, 207). Versions of a parametric stadium designed by Moretti (fig. 6) were presented as part of his *Parametric Architecture* exhibition at the Twelfth Milan Triennial

2 Gerber claims Ruiter's paper was published in *Advances in Computer Graphics III* (1988). When I looked at this book, none of the papers were titled *Parametric Design* and none of the papers were written by Ruiter (he was the editor not writer). As best I can tell, there never was a paper titled *Parametric Design* produced in 1988. The first reference I can find to Ruiter's supposed paper is in the bibliography of Javier Monedero's 1997 paper, *Parametric Design: A Review and Some Experiences*. It is unclear why Monedero included the seemingly incorrect citation since he never made reference to it in the text of his paper. As an aside: the word *parametric* does appear four times in *Advances in Computer Graphics III* – on pages 34, 218, 224, & 269 – which indicates that the use of *parametric* in relation to design was not novel at the time.

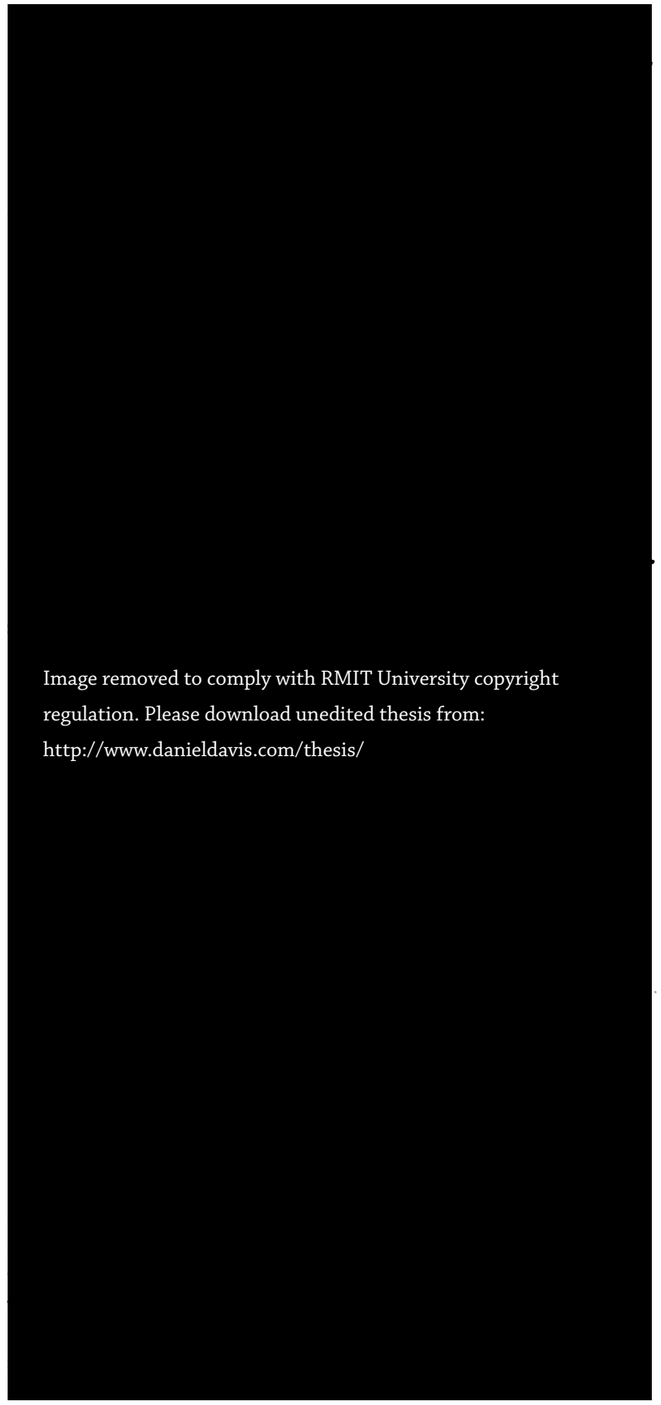
in 1960 (Bucci and Mulazzani 2000, 114). In the five years following the exhibition, between 1960 and 1965, Moretti designed the Watergate Complex, which is "believed to be the first major construction job to make significant use of computers" (Livingston 2002). The Watergate Complex is now better known for the wiretapping scandal that took place there and Moretti is "scarcely discussed" (Stiles 2006, 15) – even by the many architects who today use computers to create parametric models in the manner Moretti helped pioneer.

Moretti did not fear obscurity as much as he feared the incorrect use of mathematical terms like *parametric*. He wrote to his friend Roisecco that "inaccuracy [regarding mathematical terms] is, in truth, scarier than the ignorance before [when architects knew of neither the terms nor Moretti]" (Moretti 1971, 206). *Parametric* has a long history in mathematics and the earliest examples I can find of *parametric* being used to describe three-dimensional models comes almost one hundred years prior to Moretti's writings. One example is James Dana's 1837 paper *On the Drawing of Figures of Crystals* (other examples from the period include: Leslie 1821; Earnshaw 1839).[3] In the paper Dana explains the general steps for drawing a range of crystals and provisions for variations using language laced with parameters, variables, and ratios. For instance, in step eighteen Dana tells the reader to inscribe a parametric plane on a prism:

> If the plane to be introduced were 4P2 the parametric ratio of which is 4:2:1, we should in the same manner mark off 4 parts of e, 2 of ē and 1 of ë.
> *Dana 1837, 42*

In this quote Dana is describing the parametric relationship between three parameters of the plane (4:2:1) and the respective division of lines e, ē, and ë. The rest of the twenty-page paper possesses similar statements that explain how various parameters filter through long equations to affect the

Figure 7: Instances of James Dana's crystal drawings. *Above:* Setting up the coordinate system (Dana 1837, 41). *Below:* Impact of changing the edge chamfer ratio (Dana 1837, 43).



3 By searching for *parametric* in Google Ngrams (http://books.google.com/ngrams/) I was able to find the earliest occurrences of *parametric* from the collection of books that Google has scanned. While James Dana (1837) is one of the more compelling results, other examples include: Samuel Earnshaw (1839, 102), who wrote about "hyperbolic parametric surfaces" deformed by lines of force in a paper that gave rise to Earnshaw's theorem; and Sir John Leslie (1821, 390), who proved the self-similarity of catenary curves using "parametric circles" in his book on geometric analysis. Google has scanned only a limited collection of books so there may be even earlier examples that were not returned in these searches. Nevertheless, Dana's writings in 1837 significantly predate any claims I have found in various histories of parametric design as to the first use of the term *parametric* in relation to drawing.

drawing of assorted crystals. Dana's crystal equations resemble those that would be used by architects 175 years later to develop parametric models of buildings, engendering them with what Moretti (1957, 184) has called (incidentally) a "crystalline splendour."

Parametric is given no special significance in Dana's writing. Dana does not describe his drawings as parametric, nor does he claim, as Schumacher (2009a, 15) later would, that designing with parametric equations "justifies the enunciation of a new style in the sense of an epochal phenomenon." Rather, Dana uses parametric in its original mathematical sense, a word given no more emphasis than other technical terms like *parallel*, *intersection*, and *plane*.

When used by Dana in 1837, or by mathematicians today, *parametric* signifies what the *Concise Encyclopedia of Mathematics* calls a "set of equations that express a set of quantities as explicit functions of a number of independent variables, known as 'parameters'" (Weisstein 2003, 2150).[4] This definition sets forth two critical criteria:

1. A parametric equation expresses "a set of quantities" with a number of parameters[5].
2. The outcomes (the set of quantities) are related to the parameters through "explicit functions"[6]. This is an important point of contention in later definitions since some contemporary architects suggest that correlations constitute parametric relationships.

4 This definition is consistent with definitions in other mathematical dictionaries and encyclopedias. I have chosen to cite from the *Concise Encyclopedia of Mathematics* as the editor, Eric Weisstein (who is also the chief editor of Wolfram Mathworld) is considered an authoritative source.

5 *Parameter* can have a number of meanings, even when used by mathematicians. The grammarian James Kilpatrick (1984, 211-12) quotes a letter he received from R. E. Shipley: "With no apparent rationale, nor even a hint of reasonable extension of its use in mathematics, *parameter* has been manifestly bastardized, or worse yet, wordnapped into having meanings of consideration, factor, variable, influence, interaction, amount, measurement, quantity, quality, property, cause, effect, modification, alteration, computation etc., etc. The word has come to be endowed with 'multi-ambiguous non-specificity'." In the *Concise Encyclopedia of Mathematics* (Weisstein 2003, 2150), the term *parameter* used in the context of a parametric equation means an "independent variable." That is, a variable whose value does not depend on any other part of the equation (the prefix *para-* being Greek for *beside* or *subsidiary*).

6 An *explicit function* is a function whose output value is given explicitly in terms of independent variables. For example, the equation $x \cdot x + y \cdot y = 1$ is the implicit function for a circle. The function is implicit since the outputs (x and y) are defined in terms of one another. To make the function explicit, x and y have been defined in terms of an independent variable. Thus, the explicit function of a circle becomes: $x = \cos(t)$, $y = \sin(t)$. By a similar token, saying that 'x is roughly twice as large as t' is not an explicit function since there is ambiguity regarding the exact relationship between the variables t and x (the relationship is non-explicit).

An example of a parametric equation is the formulae that define a catenary curve:

$$x(a,t) = t$$
$$y(a,t) = a \cosh\left(\frac{t}{a}\right)$$

These two formulae meet the criterion of a parametric equation. Firstly, they express a set of quantities (in this case an x quantity and a y quantity) in terms of a number of parameters (a, which controls the shape of the curve; and t, which controls where along the curve the point occurs). Secondly, the outcomes (x & y) are related to the parameters (a & t) through explicit functions (there is no ambiguity in the relationships between these variables). This is the origin of the term *parametric*: a set of quantities expressed as an explicit function of a number of parameters.

All Design is Parametric

Since Dana's (1837) parametric crystal drawings 175 years ago, architects have gradually begun using both parametric models and the term *parametric*.[7] Early examples include Antoni Gaudí using a hanging chain model to derive the form of Colònia Güell at the turn of the twentieth-century[8] (M. Burry 2011, 231) and Frei Otto similarly using physical parametric models as a form finding technique beginning in the 1950s (Otto and Rasch 1996). Slightly after Moretti held his *Parametric Architecture* exhibition in 1960 (Bucci and Mulazzani 2000, 114), Ivan Sutherland (1963) created the first parametric software, Sketchpad. However, it was not until Parametric Technology Corporation released Pro/ENGINEER in 1988 that parametric modelling software became commercially viable (Weisberg 2008, 16.10), and it took at least another decade for parametric modelling software to be specifically designed for architects. Today architects craft parametric models in a range of software

7 I have elected not to write a complete history of parametric modelling since doing so would not contribute significantly to the argument developed in the remainder of this thesis. For those interested, I recommend Weisberg's (2008) detailed account of early CAD software and Gerber's (2007) chapter on precedents to parametric practice.

8 A hanging chain has at least four parameters: its length, its weight, and the two points it is attached to. Left to hang under the force of gravity, the chain makes a curved shape. This curve is an explicit function of the chain's parameters with the added property that when inverted the curve stands in pure compression. While there is no computer, the hanging chain is a parametric model due to the presence of parameters that control a shape derived from an explicit function (in this case calculated by gravity).

environments: from history-based modellers[9], to visual scripts[10], physical modelling[11], and textual programming environments[12].

While one could argue that architects have spent decades gradually adopting parametric modelling, some have argued that architects have always produced parametric models since all design, by definition, derives from parameters. This claim has been put forward by several authors, including David Gerber in his doctoral thesis on *Parametric Practices* where he contends:

> It must be stated that architectural design is inherently a 'parametric' process, and that the architect has always operated in a 'parametric fashion'.
> *Gerber 2007, 54*

The same argument has been made by Robert Aish and Robert Woodbury:

> Parametric modelling is not new: building components have been adapted to context for centuries.
> *Aish and Woodbury 2005, 152*

In a similar vein, Mark Burry rhetorically asks whether the opposite is true, whether non-parametric design exists:

> 'Parametric design' is tantamount to a *sine qua non*; what exactly is non-parametric design?
> *M. Burry 2011, 18*

Roland Hudson holds a similar opinion and opens his doctoral thesis, *Strategies for Parametric Design in Architecture*, with the sentence:

> This thesis begins with the assertion that all design is parametric.
> *Hudson 2010, 18*

9 History-based modellers track how the designer creates geometry, allowing the designer to make changes later. Examples include: CATIA, SolidWorks, and Pro/Engineer.
10 Visual scripts resemble flowcharts explaining how parameters generate geometry. Designers can manipulate the script's inputs or the script itself to change the model. Examples include: Grasshopper, GenerativeComponents, and Houdini.
11 Physical models like Gaudí's hanging chain model and Frei Otto's soap films use physical properties to calculate forms based on a set of parameters.
12 There are scripting interfaces included with most CAD programs. These allow designers to setup parameters and a set of explicit functions that generate geometry and other parametric outputs.

For each of these authors, the claim that 'all design is parametric' stems from the observation that all design necessarily involves parameters like budget, site, and material properties. While this is undoubtedly true, the pivotal part of a parametric equation is not the presence of parameters but rather that these parameters relate to outcomes through explicit functions. This explicit connection does not exist for all the parameters involved in a design project. Typically relationships between parameters and outcomes are correlations; the budget has a noticeable affect on the design outcome but normally the mechanism that links the budget to the outcome is – at best – ambiguous. Therefore, by interpreting *parametric* to mean, literally, *design from parameters* these authors downplay the importance of explicit relationships to parametric modelling and instead base their definition of *parametric* upon the observable interface to the model.

Change is Parametric

Another observable characteristic of a parametric model – besides the presence of parameters – is that the geometry changes when the parameters change. This leads some to claim that *change is parametric*. Chris Yessios, the founder and CEO of the modelling software FormZ, summarises the history of this interpretation:

> Initially, a parametric definition was simply a mathematical formula that required values to be substituted for a few parameters in order to generate variations from within a family of entities. Today it is used to imply that the entity once generated can easily be changed.
> *Yessios 2003, 263*

Yessios (2003, 263) acknowledges the mathematical origins of parametric modelling but also advances a definition couched in behavioural terms: the trademark behaviour of a parametric model being that it "can easily be changed." Robert Woodbury (2010, 7) seems to advance a similar definition, beginning *Elements of Parametric Design* with the two sentences: "Design is change. Parametric modelling represents change." This is followed shortly thereafter with the claim, "parametric modelling introduces fundamental change: 'marks', that is, *parts of the design*, relate and change together in

a coordinated way" (Woodbury 2010, 11).[13] Robert Aish (2011, 23) has similarly emphasised the importance of variation by saying a parametric model "directly exposes the abstract idea of geometric 'transformation'." Revit Technology Corporation[14] used a similar definition in a greeting to visitors of the Revit website:

> Para.me.tric **adj.** *Math*. A quantity or constant whose value varies with the circumstances of its application, as the radius line of a group of concentric circles, which [sic] varies with the circle under consideration. *Revit Technology Corporation 2000b* (emphasis theirs)

While Revit Technology Corporation claim that their definition comes from mathematics, the definition in no way resembles the actual mathematical definitions I cited earlier. Critically, their definition overlooks the role of explicit functions in a parametric model, an oversight also present in the various definitions given by Woodbury, Aish, and Yessios. In place of explicit functions are notions that parametric models can be defined by the variation they produce. Change is an easily identifiable characteristic of a parametric model and one that many authors choose to define parametric modelling by.

Defining parametric modelling in terms of change conjures Heraclitus's dictum 'Nothing endures but change'. Although parametric models change, so too does practically everything else in the world, except perhaps change itself. Even explicit geometric models can commonly be changed through rotation, or scaling, or moving a mesh vertex. And more specialised representations, like BIM, are set up to ensure changes to the underlying database also change the associated models. Thus, while parametric models change, and while parametric models are celebrated for being able to change, change is hardly a unique feature of parametric modelling. By saying parametric modelling is change, the various authors once again focus on what parametric models do, without considering the unique qualities of how parametric models are created.

13 Woodbury's definition nods to Sutherland's (1963, 22) explanation of Sketchpad's behaviour, "change of a model's critical part will automatically result in appropriate changes to related parts." Of course, Sutherland was not explaining the meaning of *parametric* but rather explaining Sketchpad to an audience who had never seen a person interact with a computer.

14 Revit Technology Corporation was founded by former employees of Parametric Technology Corporation. Their initial ambition was to create the "first parametric building modeler for architects and building design professionals" (RTC 2000a) although since their acquisition by Autodesk in 2002 they have begun branding what they do as Building Information Modelling (BIM).

Tooling is Parametric

Mark Burry (2011, 8) begins *Scripting Cultures* by saying, "we are moving rapidly from an era of being aspiring expert users to one of being adept digital toolmakers." Many other prominent authors describe themselves as toolmakers and claim that parametric models are a type of drawing tool (examples in key books and doctoral theses include: Aranda and Lasch 2005; M. Burry 2011; Fischer 2008; Gerber 2007; Hudson 2010; Kilian 2006; Woodbury 2010; Shelden 2002). This toolmaking analogy has been in use since at least 1983 when the then co-founder of Autodesk, John Walker (1983), made the heady charge that their actions over the coming year "will decide whether AutoCAD becomes synonymous with 'drawing tool'." In doing so Walker attempted to position AutoCAD alongside analogue drawing tools like the tee-square and the drafting table, a task he and his competitors were largely successful at. In recent years, the term has been further catalysed by Benjamin Aranda and Chris Lasch's book *Tooling* where they explain seven basic parametric recipes for what they call drawing tools.

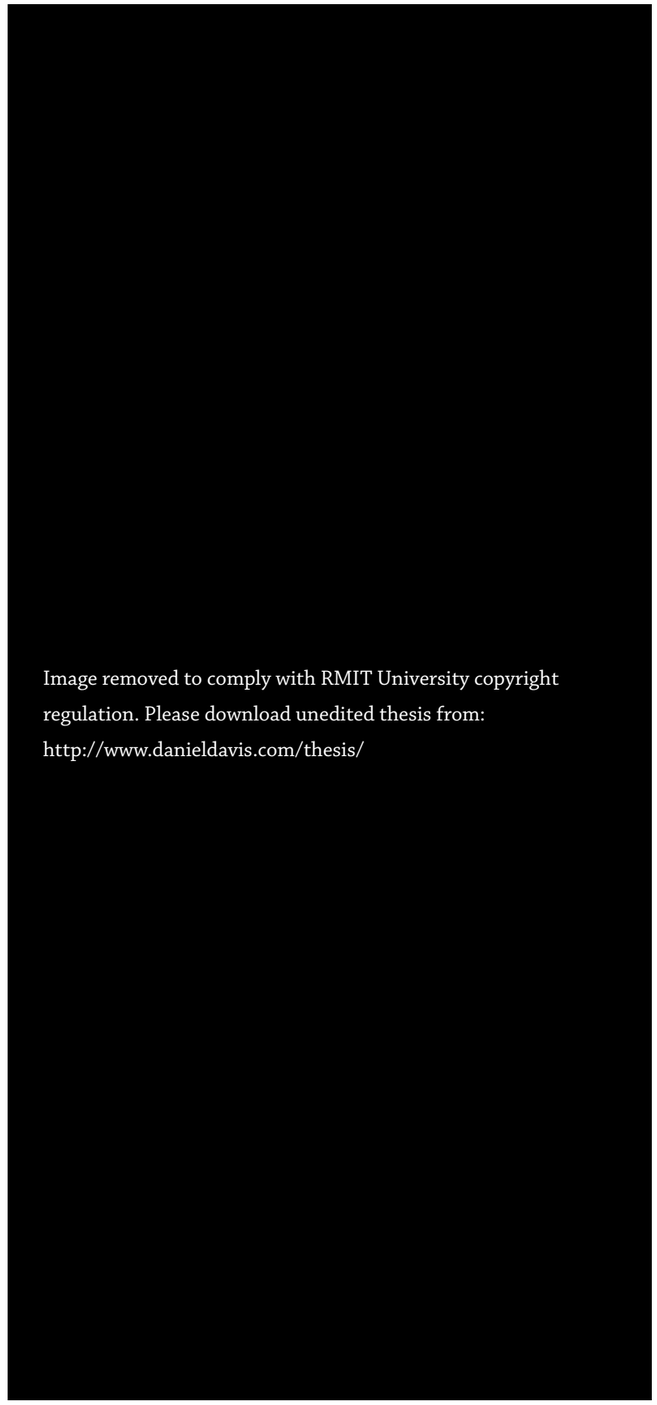
Whether tooling is an appropriate descriptor for what architects do is a question I will leave for the discussion at the end of this thesis. For now I would like to pause and consider how *tooling* implies an answer to the question *what is parametric modelling?*

The term *tooling* conveys a separation between maker and user; between the nameless person who makes a tee-square and the designer that uses the tool. Aranda and Lasch (2005, 9) reinforce this division, concluding the introduction to *Tooling* by saying, "once this field [meaning the tool] is defined as a flexible and open space, the job of designing begins." Aish (2001, 23) similarly divides the act of creating a tool and the job of designing when he remarks: "Software developers do not design buildings. Their role is to design the tools that other creative designers, architects and engineers use to design buildings." The implied division between tool use and tool making is significant: it suggests the creation and the use of a parametric model is temporally separated, and perhaps even organisationally separated.

The implications of this separation are eloquently (if unintentionally) captured by Roland Hudson (2010) in his thesis *Strategies for Parametric Design in Architecture*. Hudson (2010) draws upon many of the same authors quoted in this chapter, dividing them within his literature review under the headings "creating the model" and "exploring the design space" (fig. 8). This division continues as Hudson discusses six case studies of projects employing parametric models, talking about each parametric model exclusively under the heading "overview of the completed model," as if creating the parametric model is separate and less relevant than using the model. Hudson (2010, 230-45) then concludes his research by saying that parametric model creation and design investigation are two separate activities. Hudson (2010, 245) says that a person using a parametric model to design ends up "refining parameter values, problem descriptions and the structure of the parametric model rather than suggesting substantial changes". Given the categorical division underlying Hudson's research, it is hard to see how he could conclude anything else; a researcher is not going to see substantial changes if they only look at "overviews of the completed model".

Hudson's reasoning is not abnormal. Definitions presented earlier – that all design is parametric or that change is parametric – show how designers can become fixated on what 'completed' parametric models do, often leaving out details of how parametric models are created or changed. This bias can create the impression of a separation between a parametric model's creation and use; a separation that privileges design exploration through parameter manipulation and underplays the possibility of ongoing model development; a separation that leads Hudson and many others to say tooling is parametric.

Figure 8: A selection from the contents of Roland Hudson's (2010) thesis *Strategies for Parametric Design in Architecture*. Hudson's distinction between creating and using a parametric model comes through in his thesis structure: the literature review is split between alternating headings of "creating the model" and "exploring the design space"; and each case study evaluation focuses on the "overview of the completed model" often without discussing any aspect of the model's creation.



Parametricism is Parametric

When Owen Hatherley (2010) talks about *arrivistes* repurposing the term *parametric*, Hatherley is really talking about Patrik Schumacher (director of Zaha Hadid Architects). Parametric was adopted by Schumacher as a call to arms in his newly declared "style war" (2010) first presented as a "parametricist manifesto" (2008) based upon "parametric paradigma" (2008) and dubbed "Parametricism" (2008) at 11th Venice Architecture Biennale. Since then, the polemic has been refined and republished in countless locations, always generating much discussion.

Parametricism is a knowingly provocative notion, claiming "post-modernism and deconstructivism were mere transitional episodes" (Schumacher 2010, 43) and that parametric design will be "the great new style after modernism" (2010, 43). Schumacher (2009a, 16) identifies parametricism with a set of "negative heuristics" like "avoid rigid geometric primitives" and "avoid juxtaposition." He counterbalances this with a set of "positive heuristics" including "consider all forms to be parametrically malleable" and "differentiate gradually (at varying rates)" (Schumacher 2009a, 16).

Schumacher (2009b) illustrates his parametricism heuristics almost exclusively with Zaha Hadid projects. When I pressed Schumacher on his lack references to other projects, Schumacher – who holds a PhD in philosophy – said "I am a practicing architect before I am a theorist" (Davis 2010). By this Schumacher does not mean that he constructs parametric models as practicing architect. Schumacher never writes about using a parametric model and I can find no evidence that Schumacher creates parametric models at Zaha Hadid.[15] Rather, Schumacher's practice largely consists of reviewing what other architects have produced with parametric models. Considering Schumacher's perspective, it is somewhat understandable that he would say "the emergence of a new epochal style … is more important than methodological and procedural innovations via specific computational techniques" (Leach and Schumacher 2012). After all, it is the stylistic outputs that Schumacher sees, not the methodology or procedure. In this sense Schumacher is not too far removed from the many other theorists who also define parametric modelling in terms of what

15 I have spoken informally to a number of people who work there.

the model does. In answering the question *what is parametric modelling?* parametricism represents an extreme position, and a position many architects like to distance themselves from, yet it is a position many others come close to in their outcome focused definitions of parametric modelling.

Modelling versus Design

In contrast to Schumacher, my thesis focuses on the methodological and procedural innovations required for parametric modelling instead of focusing on enunciating the emergence of a new epochal style of parametric design. The discourse surrounding parametric design (whether *parametric design* is taken as a verb to describe the process of designing with a parametric model, or as a noun to describe the outcomes of this process) seems unlikely to reach a resolution in the near future, nor does it need to. I will leave it for others to debate how parametric design fits into the broader culture of architecture; for now there is a pressing need to understand the technological challenges presented by parametric models. Finding ways to make a parametric model more flexible may have ramifications in terms of architectural design but the immediate ramifications will be for the multitudes of architects currently using parametric models in their practice. Thus, my research focuses almost entirely on parametric modelling and leaves aside the debates surrounding the design implications.

What is a Parametric Model?

The definition of a parametric model, like a parametric model itself, has an unsettled variability. At any particular time *parametric* may signify all of design, or only the designs that change, or tooling, or design in the style of parametricism. This collective disagreement exists even on an individual level, with many prominent authors providing different definitions across the span of their work (in the proceeding pages, I have often been able to quote the same author under different definitions of parametric). Unsurprisingly, architects like Patrik Schumacher have seized this confusion as an opportunity claim the meaning of parametric, whilst others have distanced themselves from the term altogether. At SIAL[16] for instance, parametric models are often referred to as *flexible models*

16 The Spatial Information Architecture Laboratory (SIAL) is a research unit within the Royal Melbourne Institute of Technology (RMIT). My PhD is part of SIAL's project *Challenging the Flexibility of the Inflexible Digital Model* – a title that deliberately uses *flexible model* instead of *parametric model*.

(M. Burry 2011, 105), which is a description that emphasises – like most definitions of *parametric* – what the models do rather than how the models were created.

The creation of the parametric model distinguishes it from other forms of architectural representation. Returning to the Concise Encyclopedia of Mathematics, a *parametric equation* is defined as a "set of equations that express a set of quantities as explicit functions of a number of independent variables, known as 'parameters'" (Weisstein 2003, 2150). The mathematical definition can be refined by recognising that the "set of quantities" in the context of design representation is typically geometry (although not always). Thus, a parametric model can be defined as: a set of equations that express a geometric model as explicit functions of a number of parameters. This is the intended meaning when nineteenth-century scientists and mathematicians like James Dana (1837) refer to parts of their geometric drawings as parametric. This is what mathematician Samuel Geisberg (Teresko 1993, 28) meant when he founded Parametric Technology Corporation and created the first commercially successful parametric software. This is the definition used by Fabian Scheurer and Hanno Stehling (2011, 75) as well as Ipek Dino (2012, 208-10). And when Woodbury (2010, 11-22) describes the mechanics of a forward propagating parametric model in his chapter "What is Parametric Modelling?" the model he describes conforms to this definition. Therefore, a parametric model is unique, not because it has parameters (all design, by definition, has parameters), not because it changes (other design representations change), not because it is a tool or a style of architecture, a parametric model is unique not for what it does but rather for how it was created. A parametric model is created by a designer explicitly stating *how* outcomes derive from a set of parameters.

The explicit connection between parameters and the geometric model potentially excludes a number of model types. Dino (2012, 209) has argued linguistic algorithms (such as shape grammars) and biological algorithms (such as genetic algorithms, flocking, and cellular automata) tend not to be parametric because they lack explicit connections. While these algorithms may contain parameters, their parameters work like a budget in a brief; they undoubtably influence the outcome but there is no explicit connection between a specific parameter and a specific outcome. Yet the boundary between parametric and non-parametric is not clear

cut. For instance, Sketchpad (Sutherland 1963, 110-19) has two solving methods: the one-pass method, which analytically solves the explicit functions (Sutherland 1963, 118-19); and the relaxation method, which bypasses the explicit functions through numeric optimisation. Sketchpad seamlessly switches between the two solving methods and to an end user they both appear parametric even though one relies upon explicit functions while the other does not. Other fringe cases include BIM models where changes to data may trigger a set of functions that recalculate a series of models. Even explicit geometry has some parametric characteristics. For instance, the endpoint of a line could be thought of as a parameter to a set of functions that transform the line. While I am aware of these grey areas, for the remainder of this thesis I will be discussing models that are unambiguously parametric – models where the designer has defined the explicit connections between parameters and the geometry. In the next section I consider why a designer would want to do so.

2.2 Why Use a Parametric Model?

If you were to ask an architect to describe a medium for designing architecture, one that fosters creativity and exploration, they would probably not reply 'a set of equations that express a geometric model as explicit functions of a number of parameters'. Yet explicit functions and parameters are the medium of choice for the many architects who design with parametric models. Understanding why architects choose to use parametric models – a seemingly counterintuitive medium for creativity and exploration – is a crucial step towards understanding the challenges associated with parametric modelling.

Thinking Parametrically

Expressing design intentions with parameters and explicit functions requires a different way of thinking than most designers are accustomed to. In addition to thinking about what they are designing, architects working with parametric models must also think about the logical sequence of formulas, parameters, and relationships that explain how to create their

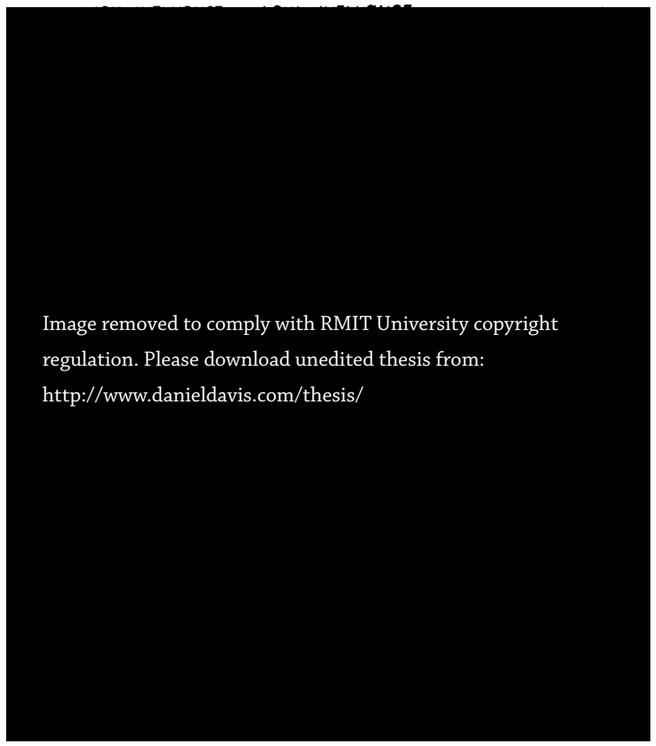


Figure 9: Paulson's curve (1976, 588). In the text accompanying this graph, Paulson talks about the benefits of making early decisions when the designer's level of influence is high.

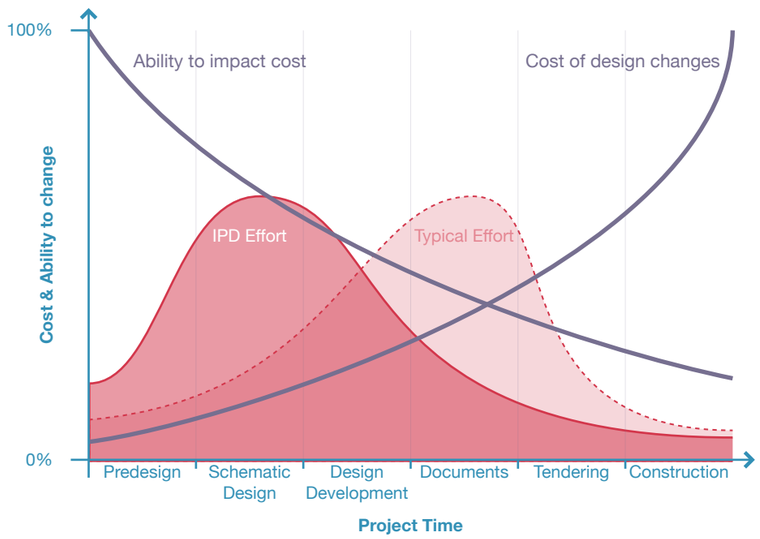


Figure 10: MacLeamy's curve (2001). MacLeamy advocates taking the typical design effort and shifting it to an earlier stage of the project. In theory this means that designers are working when their decisions have the most impact and least associated cost.

designs (Aish 2005, 12; Woodbury 2010 24-25). Some dub this *parametric thinking* or *algorithmic thinking*. Learning to think parametrically is "a hard-won skill, not acquired with ease" say all but one of forty experts interviewed by Mark Burry (2011, 38). Although learning to design in such a mediated manner can be difficult, the logical precision can also be enjoyable for designers who relish pushing back against imposed constraints, and for designers who like how parametric modelling forces them to explicitly state (and therefore consider) every geometric relationship (Aish 2005, 12; M. Burry 2011, 38-39; Kilian 2006, 300-03; Woodbury 2010 24-25). However, the real benefit of learning to think parametrically comes from the cost of design changes.

The Cost of Change

In 1976 Boyd Paulson sketched a graph (fig. 9) showing that a designer's level of influence over an architecture project decreases as the project progresses. Paulson (1976, 588) points out that the first decision a designer faces on any project – whether to commence the project or not – has total influence over the project's future. He goes on to argue that all subsequent decisions have a diminishing influence and are generally more costly to implement. In other words: as designs become more developed, they also become more difficult to change. Paulson published his observations in a few construction management textbooks (Barrie and Paulson 1991) but the idea never became widely circulated.

Paulson's graph reappeared in May 2001 at a resort in Mexico. The leaders from HOK (one of the world's largest architecture firms) had gathered at the resort to discuss "key strategies for the future" (HOK 2012). During the discussions, Patrick MacLeamy presented a graph (fig. 10) showing that a designer has the most "ability to impact [a project's] cost and functional capabilities" (MacLeamy 2010) at the start a project, and that this ability decreases during a project while the cost of making design changes increases. The graph MacLeamy presented was identical to Paulson's. MacLeamy claimed the work as his own (perhaps unaware of Paulson's efforts) and HOK went on to promote Paulson's graph under the name *MacLeamy's curve* (HOK 2012) – a name that has stuck thanks in part to

HOK's marketing clout.[17] Two years after MacLeamy presented the graph to the leaders of HOK, MacLeamy was appointed CEO, a position he has held for almost a decade (HOK 2012).

As CEO of HOK and the international chair of buildingSMART[18], MacLeamy has used his curve to champion the *front-loading* of architecture projects. MacLeamy (2010) advocates making design decisions early in the project (shifting the design effort forward) since his curve shows that design changes are less costly to make at the start of the project compared to the end. Paulson (1976, 591-92) drew the same conclusions from his graph and suggested construction knowledge should be injected earlier in the design process. More sophisticated examples of front-loading are given by MacLeamy (2010) who advocates both of the following: Integrated Project Delivery (IPD), which contractually amalgamates all the project parties to guide the design team towards viable solutions early in the project; and BIM, which provides a central project database to improve communication between team-members while also aiding early stage simulations and later stage project documentation. These ideas have been widely disseminated and MacLeamy's conception of front-loading has informed contemporary architectural practice in everything from the American Institute of Architects (AIA 2007, 21-31) guidelines for IPD to the instruction manuals for Autodesk's Revit (Read, Vandezande, and Krygiel 2012, fig. 5.7).

Discussing the *cost of change* may make some designers uncomfortable, particularly if they perceive their *costly* changes as valuable contributions to a project. However, cost in this context is a measure of the designer's capacity to make change; the designer's ability to design. Ostensibly, front-loading should empower designers by encouraging them to act when the cost of change is low and their capacity to make change is high. Yet, the paradox of front-loading is that by forcing design decisions early in the project, the project becomes more developed and therefore, according to MacLeamy's curve, more costly to change later on. It is this increase in the cost of change that should make designers uncomfortable because it signals a reduction in the designer's capacity to make late changes.

17 I am extremely grateful to Noel Carpenter for drawing my attention to Paulson's work in a comment Carpenter left on a blogpost I wrote about MacLeamy (Davis 2011a). As best I can tell, no previous research has cited Paulson when discussing MacLeamy's curve.

18 BuildingSMART is an influential consortium of CAD manufactures and users that develops open standards for describing buildings. They are perhaps best known for the development of the IFC standard, which facilitates interoperability in between BIM software.

The introduction of parametric modelling was motivated by a desire to decrease the cost of change. This motivation is discussed by Samuel Geisberg, the founder of Parametric Technology Corporation, during an interview with *Industry Week* in 1993:

> The goal is to create a system that would be flexible enough to encourage the engineer to easily consider a variety of designs. And the cost of making design changes ought to be as close to zero as possible. In addition, the traditional CAD/CAM software of the time unrealistically restricted low-cost changes to only the very front end of the design-engineering process.
> *Geisberg quoted in: Teresko 1993, 28*

Geisberg's comments suggest that instead of looking at MacLeamy's cost of change curve and concluding design efforts should be shifted to the "very front end of the design-engineering process," a better conclusion may be to shift the cost of change curve so that the "cost of making design changes [is] as close to zero as possible." In theory, a parametric model helps lower the cost of change provided the model's parameters and explicit functions require less effort to change than alternative modelling methods. Geisberg calls this *flexibility* (Teresko 1993, 28). In chapter 4 I discuss the various nuances of flexibility, but for now flexibility can be understood as a measure of the cost of design changes and, by proxy, a component of the designer's capacity to design.

Flexibility makes-up the central tenet of parametric modelling. By maintaining a flexible model the designer can afford to make changes, which is important given the inevitability of change on an architecture project. While some changes can be anticipated and perhaps even front-loaded, many changes come from forces outside the designer's sphere of influence. For instance, the client can change the brief, politicians can change the legislation, and market forces can change the price of materials. Other changes occur because design is a knowledge generating process. Often it is only through iteration, exploration, and reflection that the problem – much less the design response – becomes known (Glegg 1969; Schön 1983; Lawson 2005; Cross 2006). In the face of these inevitable changes, the flexibility of a parametric model's parameters and explicit functions makes for an alluring design medium; one many architects employ to help improve the designer's capacity to design.

2.3 Reported Difficulties with Models in Practice

While the flexibility of a parametric model purportedly helps designers accommodate change, there is growing evidence that this is not always the case. "Many times," writes Rick Smith (2007, 2), architects working with parametric models are finding they have to "start all over again" once changes incapacitate their models. Parametric Technology Corporation (PTC 2008, 1) admit "this situation is fairly common" with users often finding that they spend "too much time re-creating designs, or can't respond to unexpected changes fast enough, or [that their] design cycles are actually taking longer [compared to using a non-parametric model]" (2008, 1). A similar sentiment is expressed by the authors quoted at the very start of this thesis (Gerber 2007, 205; J. Burry 2007, 622; Holzer et al. 2007, 639; Aish and Woodbury 2005, 151; M. Burry 1996, 78). In the following section I revisit what these authors say about the practice of parametric modelling and investigate the associated challenges they reveal.

Evidence of Challenges

Very few architects have spoken publicly about how they construct and maintain their parametric models; fewer still in a critical manner. This is not entirely surprising considering the relatively recent adoption of parametric modelling by most architects. Only in the past decade have the challenges of computational power, workflows, and algorithms receded to the point where parametric modelling has gone from an issue largely of theory to the subject of practice.

When architects do write about the practice parametric modelling there is a tendency to understate the challenges. As Thomas Fischer (2008, 245) laments, firsthand accounts of "failures and dead-ends … seem to be rare and overshadowed by the great number of post-rationalised, outcome-focused reports on digital design toolmaking." This observation ties into the point made earlier in this chapter: architects are inclined to focus more on what parametric models do than how the models come to be. From this perspective, the failures and dead-ends can be hard to see. But through the veneer of architects talking positively about the outcomes of projects they were personally involved in, there are fragments containing frank admissions of the problems encountered. The handful of authors who have written candidly about these challenges make up the bibliography of this section.

The most explicit critique of parametric modelling comes in a short, six-page paper entitled *Technical Notes from experiences and studies in using Parametric and BIM architectural software* (Smith 2007). The paper is not peer reviewed, has not been published, and lists only one source. Ordinarily such a paper could be dismissed as misattributed opinion, only, this paper is written by Rick Smith.

Rick Smith played a large part in introducing parametric modelling to the architecture industry. Smith began working as a CAD technician for Lockheed in 1979 (Smith 2010), well before most architecture firms had computers. By the start of the 1990s, Smith was designing parts for the United States space shuttle with Dassault Systèmes' CATIA (Smith 2010). Based on Smith's experience in the aerospace industry, Frank Gehry and Associates hired Smith in 1991 to help design the Barcelona Fish using CATIA – one of the first times software of this calibre was used in the architecture industry. Smith ended up spending a decade consulting to Gehry, employing parametric modelling on some of Gehry's most prominent projects, such as the Guggenheim Bilbao (1993-97), the Experience Music Project (1995-00), and the Walt Disney Concert Hall (1992-03). The success of this collaboration helped spawn the sister company Gehry Technology (incorporated in 2001), which went on to develop the parametric modelling software Digital Project (2004) – a modified version of CATIA intended for architects. Given the decades Smith has spent helping pioneer parametric modelling in the architecture industry, it is very significant for him to now turn around and highlight the flaws.

Since 2007 Smith has announced the challenges associated with parametric modelling in the short, six-page white-paper prominently displayed on the website of his consultancy, Virtual Build Technologies. The white-paper identifies five major shortcomings with parametric modelling (Smith 2007, 2):

1. Parametric models require a degree of front-loading.
2. Anticipating flexibility can be difficult.
3. Major changes break parametric models.
4. Changes can be hard to see visually.
5. Reusing and sharing models is problematic.

Smith's points seem to resonate with what other authors have written, even if they do not write so emphatically. In the following section I take in turn each of Smiths five critiques and consider whether evidence from peer-reviewed authors corroborates his self-published opinions.

One: Front-loading

> When you model using parametrics you are programming following similar logic and procedural steps as you would in software programming. You first have to conceptualize what it is you're going to model in advance and its logic. You then program, debug and test all the possible ramifications where the parametric program might fail. In doing so you may over constrain or find that you need to adjust the program or begin programming all over again because you have taken the wrong approach.
> *Smith 2007, 2*

In Smith's first critique of parametric modelling, he points out that creating a parametric model requires some degree of upfront planning. This is reiterated by Weisberg (2008, 16.12) who recalls that even in 1993 designers creating parametric models in Pro/ENGINEER needed to "carefully plan the design, defining ahead of time which major elements would be dependent upon other elements." Planning is a necessary component of parametric modelling because the logical rigidity of a model's explicit functions requires that the designer anticipate, to some degree, the parameters of the model and the hierarchy of dependencies between functions. This "prerationalization process is often found to be

arduous," states Gerber (2007, 205), "as it requires a significant amount of upfront cognitive investment." While pre-rationalisation can be onerous, the real difficulty of pre-rationalisation is not the upfront cognitive investment but rather the risk that the designer may invest time on "the wrong approach" (Smith 2007, 2). As Axel Kilian (2006, 54) warns, "structuring the design approach early on in the design process … offers little flexibility once a model has been created." This is a similar problem to MacLeamy's front-loading: many changes in an architecture project cannot be anticipated upfront, and decisions made too early in the project may raise the subsequent cost of design changes since any major change will undo all of the initial work. Aish and Woodbury (2005, 151) echo this statement by acknowledging "parameterization may require additional effort, may increase complexity of local design decisions and increase the number of items to which attention must be paid in task completion." The additional effort required to design using explicit functions necessitates that designers have some notion of the design outcome prior to modelling. This upfront planning can be challenging, particularly in a process as notoriously hard to anticipate as the design process.

Two: Anticipating Flexibility

> Once you think you have a working parametric model you may still find you haven't programmed a parameter of the geometry in a way that is adjustable to a designer's future request. A designer might say I want to move and twist this wall, but you did not foresee that move and there is no parameter to accommodate the change. It then unravels your program. Many times you will have to start all over again. Imagine trying to do this on a complex and fully integrated building.
> *Smith 2007, 2*

Part of the upfront planning of a parametric model, according to Smith's second critique, involves anticipating future design changes. If changes can be anticipated, the model can be structured with the appropriate parameters to accommodate these changes. However, if a change is not anticipated, the designer must accommodate the change by modifying the model's explicit functions. This process of "conceiving, arranging and

editing dependencies is the key parametric task" according to Woodbury (2010, 25). As with the task of initially building the parametric model, modifying the explicit functions can be challenging, particularly on a "complex and fully integrated building" (Smith 2007, 2). A critical endeavour of the designer is therefore to avoid unnecessarily rebuilding the model by anticipating future changes and creating parametric models with the flexibility to accommodate these anticipated changes from the start.

Given the importance of anticipating flexibility, a seemingly obvious response is to make every aspect of a model flexible; add parameters for every possible whimsy of the designer. But parameters come at a cost. They require work upfront to implement, and they require even more work to change. This investment may not pay off if the parameter is rarely used. Therefore, the skill of anticipating flexibility is getting the balance right between too much and too little flexibility. Fabian Scheurer and Hanno Stehling share this sentiment:

> The challenge of building a parametric model is to untangle the interdependencies created by different requirements and find a set of rules that is as simple as possible while remaining flexible enough to accommodate every occurring case. In other words: to pinpoint the view to the exact level of abstraction where no important point is lost and no one gets distracted by unnecessary detail.
> *Scheurer and Stehling 2011, 75*

Thus, an ideal parametric model encompasses all the variations the designer wants to explore with the most concise set of parameters possible. According to Jane Burry and Mark Burry (2006, 793) this catches designers in a paradox: upfront they need to anticipate potential changes to the model, yet they lack the knowledge to do so because it is the "very variability of the model that uncovers potential ranges of [new] possibilities that leads to design explorations." The paradox of anticipating flexibility presents a challenge for designers, one they must overcome to avoid making major model changes.

Three: Major Changes

> After all the time and effort of programming the geometry to where you think you have it right, you may find you still have to start all over again because the initial design concept has completely changed.
> *Smith 2007, 2*

Smith's third critique of parametric modelling is that major changes will break even the most flexible parametric models. Ordinarily changes can be accommodated either through modifying parameters or by modifying the model's explicit functions. However, an industry survey by the Aberdeen Group (2007, i) of more than 150 firms (8% of which were architecture firms) found that designers "often end up spending more time fixing models than if they had simply started from scratch." Weisberg (2008, 16.12) noted similar behaviour in his observations of engineers, concluding the difficulty of modifying a parametric model's explicit functions was such that "in extreme cases (and sometimes in cases that were not particularly that extreme), the user was forced to totally recreate the model." In reality the designer is never forced since a designer always has the option of not making a particular change. Although, admittedly, there is little comfort in being asked to choose between either completely rebuilding your parametric model or compromising your design intentions to fit the limitations of an existing parametric model.

Architects tend to be unwilling to talk about the "failures and dead-ends" (Fischer 2008, 245) that result in models being rebuilt. When they do, it is often only in passing. For instance, David Gerber (2007, 205) mentions that if the "topology of a project changes the [parametric] model generally needs to be remade." Yet apart from this single sentence, Gerber does not dedicate any more space in his five hundred-page thesis to the fact that models employed for their flexibility apparently fail if the topology of the project changes – a seemingly critical detail for a thesis entitled *Parametric Practices*. Slightly more depth can be found in various exposés of the parametric modelling process. For example, Mark Burry (1996, 78) speaks to the issue of topological fragility when discussing the design process for the triforium column of the Sagrada Família. Burry initially built the parametric model from hyperbolic paraboloid geometry but, during the subsequent design process, the design team decided to test whether the column could instead be made from conoid geometry.

When Burry (1996, 78) tried to make this topological transformation, he found himself in a situation where "there is no solution other than to completely disassemble the model and restart at the critical decision." These struggles with topological transformations, particularly ones that result in models being rebuilt, are especially acute for architects given the relative lack of topological consistency in building forms when compared to the typically homeomorphic forms of boats, planes, and cars dealt with in other industries.

While topological transformations can be difficult, they are not the only type of major change that causes models to be rebuilt. Dominik Holzer et al. (2007) tells of how unanticipated changes can completely disrupt a model, as happened on the model for AAMI Park stadium in Melbourne. Holzer joined the project as a member of ARUP's team assisting with the structural calculations during the design development. By this stage of the design process, many parts of the project were finalised and therefore included as invariable geometry in the model. It was expected that the design would become more resolved during the design development process, however, the opposite happened: "the number of variable design factors increased" (Holzer et al. 2007, 637). As more and more of the original design intent was changed, and as the changes impacted parts of the model initially assumed to be invariable, the changes became of "such a disruptive nature that the parametric model schema could not cope with them" and the "model consequently fell apart" (Holzer et al. 2007, 639). Holzer had no choice but to rebuild the model. Changing a parametric model is often so disruptive that Hudson (2010, 240) recommends "early models should be treated as disposable and not precious." Jane Burry (2007, 622) also notes that it is commonplace when parametric modelling to "return to the metaphorical drawing board to edit the relational graph or remodel completely." While Weisberg (2008, 16.12) observes designers trying to avoid this by "planning their design work in order to avoid having to start over if major changes were made to the design." However, even with the best front-loaded anticipation of changes, the inflexibility of parametric models in the face of major changes often present the designer with only one viable option: start over.

There will always be changes outside the designer's control – to the legislation, budget, and the client's favourite colour. And since designers learn through designing, there will always be changes that cannot be

anticipated prior to modelling. These changes are ordinarily accommodated by modifying parameters or by modifying the model's explicit functions. However, if the change is large enough, or the topology unfamiliar enough, this choice will diminish to just one option: rebuild the model. A decidedly inflexible outcome.

Four: Change Blindness

> Once you have your program working if anyone changes a parameter it could affect the geometry somewhere in the design that you didn't want to be changed. This occurs often and the change may not be detected until much later in the design phase, or even worse, in the more expensive construction phase.
> *Smith 2007, 2*

Designers often fail to observe changes in models, says Smith in his fourth critique of parametric modelling. His claim is corroborated by strong empirical evidence from numerous psychologists. Summarising this existing research, Simons and Levin (1997, 261) note, "experiments using a diverse range of methods and displays have produced strikingly similar results: unless a change to a visual scene produces a localizable change or transient at a specific position on the retina, generally, people will not detect it." This phenomenon is known as *change blindness*. A study by Nasirova et al. (2011) of twenty participants using parametric models found "change blindness did indeed occur … making change detection for 3D parametric [modelling] highly challenging, slow and confusing" (2011, 762; a similar point is made in: Erhan, Woodbury, and Salmasi 2009). A designer suffering from change blindness will essentially be unable to see certain model changes, even though the changes are unobscured on screen while the designer actively looks for them. On occasion the designer will fail to see any changes at all. Phillips (1974) demonstrates that this problem is exacerbated by the latency between seeing one variation and seeing the other variation (which, in a parametric model, is the time between making a change and seeing the result). The propensity of designers to suffer from change blindness has two major implications for the application of parametric modelling:

- Firstly, as Rich Smith warned, changes to a parametric model may have undetected consequences. In chapter 7 I discuss a project where inadvertent changes went undetected until they caused near catastrophic problems during the construction phase.
- Secondly, if a designer is unable to identify what has changed between two model variations, then they may struggle to make an informed evaluation of design changes.

Five: Reuse and Sharing

> This also points to the fact that any operator using the model needs intimate knowledge of the parametric program that is written for that specific design. This logic knowledge is not easily transferred with the 3D model. In a sense the original programmer of the model then becomes the owner of the model. Many times if the program is too complex the original programmer is the only one who can work with it. *Smith 2007, 2*

Smith's final critique of parametric modelling is that parametric models are difficult to reuse and share. Parametric Technology Corporation (PTC 2008, 3) acknowledge this problem and say "even after a model is created, other designers can't easily modify the design because they don't possess the knowledge about how it was created and the original design intent." Yanni Loukissas (2009) observes that because parametric models are difficult to share, one architect within an organisation inevitably becomes the "keeper of the geometry" – the person responsible for the parametric model by virtu of being the only person capable of modifying the model. A similar conclusion was drawn by the Aberdeen Group's survey (2007, 3), which identified the following "top four challenges to design reuse" (2007, 3):

1. Model modification requires expert CAD knowledge.
2. Models are inflexible and fail after changes.
3. Users cannot find models to reuse.
4. Only the original designer can change models successfully.

The first, second, and fourth item on this list are singled out by the Aberdeen Group (2007, 3) "as testament to the fact that feature based models [a synonym for parametric models] can be a barrier to design

reuse." In many ways, these three points are just manifestations of Smith's third critique: major changes often break parametric models. After all, if a designer breaks their own model trying to implement a change, then a designer unfamiliar with the model is going to be in a far worse position to adapt that model to a totally new circumstance. Research by Kolarić et al. (2010) indicates that even simple tasks becomes cognitively demanding when a parametric model is unfamiliar. Identifying a relationship in a parametric model containing twelve objects took participants on average ten seconds to complete, and they were wrong 20% of the time (even though they had a 50% chance of randomly answering the yes/no question correctly; Kolarić et al. 2010, 709-10). This is to say nothing of the difficulty of identifying relationships in a much larger model or the difficulty of modifying relationships once they are correctly found – two tasks necessary for reuse and sharing.

The Challenges of Parametric Modelling

Rick Smith's five critiques of parametric modelling are a subject absent from much of the architectural discourse. However, dispersed through the writings of architectural theorists, practicing architects, software manufactures, and psychologists are pieces of evidence that strongly support Smith's five claims. Collating this fragmented evidence represents a significant furthering in our understanding of parametric modelling. Perhaps most notably, it demonstrates that parametric models used in practice are often blindsided by the very thing they purportedly accommodate: change.

Smith's first three critiques (front-loading, anticipating flexibility, and major changes breaking models) are really manifestations of the same thing: the difficulty of expressing unsettled design intentions with explicit functions. Given this difficulty, front-loading is frequently necessary to orchestrate a parametric model's explicit functions into an appropriate hierarchy. Once in this hierarchy, the difficulty of changing the relationships often prompts designers to try – somewhat in vain – to avoid changes by anticipating them first. This is a situation that unfortunately shifts the rhetoric around parametric modelling, from one of designers embracing change, to one of designers eschewing change. To a lesser extent, the challenges of working with explicit functions also contribute

to the problems with sharing models, since being unfamiliar with a model only exacerbates the difficulties of changing the model. The only critique from Smith not directly linked to fragility of explicit functions is change blindness, which instead arises from difficulties all humans have in visually observing and evaluating change.

Smith's critiques show that architects are clearly facing challenges with expressing design intentions using explicit functions, while also struggling to observe design changes and to reuse models. These observations are substantiated independent of the parametric modelling software used, the design team composition, the stage parametric modelling is used in a project, the types of changes asked for, and the design complexity. While each of these circumstances may be singled out as a problem, Smith's critiques suggest that they are symptoms, or at least aggravations, of problems common to all parametric models. These are problems largely concerning explicit functions but they are not solely technological, they concern the designer and they also concern the inherent unpredictability of the design process. Smith's critiques indicate that designers are often finding themselves in situations where they cannot modify the model's explicit functions and the designer is left with two undesirable choices: they can delay the project and rebuild the model, or they can avoid making the change altogether.

2.4 Conclusion

The difficulties of parametric modelling are set in motion by the struggles to define it. The term *parametric* originates in mathematics where, since at least the 1830s, mathematicians and scientists have used the term in relation to various geometric representations. However, as architects have adopted parametric modelling as a design medium, the definition of *parametric* has become muddied. Now when architects use the term *parametric*, they could mean all of design, or only the designs that change, or tooling, or design in the style of parametricism. The disagreement exists even on an individual level, with many prominent authors defining *parametric* differently across the span of their work.

The commonality of these contended definitions is that they focus on what parametric models do. To a certain extent this makes sense: it is, after all, what parametric models do that makes them interesting to architects. For architects, parametric models purportedly improve the designer's ability to make changes, thereby improving their capacity to design. In theory a designer can modify a model's parameters and see the design change almost instantly. As such, parametric models have come to be understood in terms of their outputs; a method for producing tools, or making parametricism, or creating design representations that change in relation to parameters. This focus on what parametric models do suggests a separation between creating and doing, a separation that underplays the significance of creating and maintaining a parametric model.

It is the construction and maintenance of the explicit relationships inside a parametric model that distinguishes parametric modelling from other forms of design representation. As such, I define a parametric model as many mathematicians would: as a set of equations that express a geometric model as explicit functions of a number of parameters. While arriving at this definition has been a contribution of this chapter, the primary contribution of this chapter has been to expose the difficulties associated with using explicit functions to design. Learning to express uncertain design outcomes with the computational logic of explicit functions is, for most architects, a "hard-won skill" (M. Burry 2011, 38). Even for experienced practitioners, like Rick Smith, the networks of explicit functions they weave often become so brittle that starting over is easier than making a change. Flexible parametric models often turn out to be inflexible in practice, and models set up to embrace change often instead end up restricting change. For architects, these difficulties are largely without precedent since parametric modelling is often "more similar to programming than to conventional design" (Wesiberg 2008, 16:12). The difficulties are familiar, however, to many software engineers who also often struggle to create flexible code – as I will detail in the following chapter.

3 The Design of Software Engineering

To a computer, a parametric model reads as a set of instructions. The computer takes the inputs, applies the sequence of explicit functions, and thereby generates the parametric model's outputs. "Anybody involved in any job that ultimately creates instructions that are executed by a computer, machine or even biological entity, can be said to be programming" argues David Rutten (2012), the developer of the popular parametric modelling interface, Grasshopper. This is not to say programming and parametric modelling are synonymous. There are clearly significant differences between designing architecture and designing software. Yet in both cases, there is a common concern with automating computation through sequences of instructions. Despite this "common ground" (Woodbury 2010, 66), and despite architects recognising that parametric modelling is often "more similar to programming than to conventional design" (Weisberg 2008, 16.12), the implications of the parallels between parametric modelling and software engineering remain largely unexplored. In particular, two pertinent questions remain unaddressed: if parametric modelling and software engineering both concern the automation of computers, do they both encounter the same challenges when doing so? And if they share the same challenges, are parts of their respective bodies of knowledge transferable in alleviating these challenges?

Woodbury, Aish, and Kilian (2007) have already shown that one area of software engineering – design patterns – is applicable to the practice of parametric modelling. Yet subsequently Woodbury (2010, 9) has been cautious in suggesting that architects can learn from software engineers, saying the practices "differ in more than expertise." Woodbury (2010, 9) goes on to describe architects as "amateur programmers" who naturally "leave abstraction, generality and reuse mostly for 'real programmers'." In this chapter I will show how abstraction, generality, and reuse have not always been the foremost concern of 'real programmers', and how Woodbury's assessment of contemporary architects could equally apply

to past software engineers. Thus, while Woodbury (2010, 9) sees today's architects as amateur programmers who are largely disinterested in software engineering, past preferences need not inform future practices. Given the success Woodbury et al. (2007) have had at improving the practice of parametric modelling with knowledge from software engineering, there is reason to suspect many more parts of the software engineering body of knowledge are also applicable to the practice of parametric modelling.

In this chapter I aim to identify the areas of knowledge employed by software engineers that could potentially help architects creating flexible parametric models. I begin the chapter by discussing how programmers once faced a software crisis not too dissimilar to the challenges architects are currently facing with their parametric models. I go on to discuss the body of knowledge that helped programmers overcome the software crisis and hypothesise about which aspects of this body of knowledge may be applicable to the practice of parametric modelling.

3.1 The Software Crisis

In the 1960s, around the time that Ivan Sutherland was creating Sketchpad, a number of big software projects unexpectedly failed. These failures "brought big companies [like IBM] to the brink of collapse" recalls Turing award-winner Niklaus Wirth (2008, 33) in his *Brief History of Software Engineering*.[1] The most shocking feature of the failures is that they happened during a period of substantial progress in computation; a period where newly invented third-generation programming languages were running atop processors with exponentially increasing speeds (Wirth 2008, 33). Yet, despite these advances, projects were coming in significantly over budget, they were late or, even worse, they were abandoned. A notable example is IBM's System/360 project, managed by Frederick Brooks, which in 1964 was one of the largest software projects ever undertaken. The size of the project was possible since computers had become "several orders of magnitude more powerful" (Dijkstra 1972, 861) but the size of the project also amplified fundamental problems with programming. These problems could not be overcome by hiring several orders of magnitude more programmers. "Adding manpower to a late software project makes it later" says Brooks (1975, 25) reflecting on his management of System/360 in the seminal software engineering book, *The Mythical Man-month*. In the end, IBM's ambitious System/360 unification, like many software engineering projects in the 1960s, was years late and cost millions of dollars more than budgeted (Philipson 2005, 19).

And so began *the software crisis*, a period when the hope borne of the relentless progression of computation was crushed; not by processing speeds derailing from their unlikely exponential increases but rather "crushed by the complexities of our own making" (Dijkstra 1997, 63); crushed by the challenge of simply writing software (Dorfman and Thayer 1996, 1-3). Wirth (2008, 33) observes "it was slowly recognized that programming was a difficult task, and that mastering complex problems was non-trivial, even when – or because – computers were so powerful." This realisation resembles the current situation in architecture, where the vast improvements in parametric modelling over the past decade have exposed the difficulties of simply creating a parametric model. In much the same way architects may blame themselves for failing to anticipate changes to an inflexible parametric model, programmers feared human cognition, not computer power, would be the limiting factor in the application of computation. This idea was so alarming that in 1968 NATO assembled a team of scientists "to shed further light on the many current problems in software engineering" (Naur and Randell 1968, 14).

The NATO *Software Engineering* conference was a watershed moment. Amongst discussions of whether anyone had died from the software crisis[2] was talk of "slipped schedules, extensive rewriting, much lost effort, large numbers of bugs, and an inflexible and unwieldy product" (Naur and Randell 1968, 122). These issues describe, almost word-for-word, the challenges many architects face when using parametric models (see chap. 2.3). In responding to these difficulties, the inclination at the NATO conference was to gather data rather than rely on intuition.

1 While computers are a relatively recent invention, their rapid development has left behind an immense history. In this chapter I only touch two aspects of this history: the software crisis and the cost of change curve. For a more complete history I would recommend starting with Wirth's (2008) *Brief History of Software Engineering*, which references a number of the key papers. Unfortunately it seems no one has yet written a comprehensive book on history of software engineering – perhaps due to the size and speed of the industry – so beyond Wirth's paper the best sources tend to be books and articles published from the period, such as Brook's (1975) *The Mythical Man-month*. Numerous guides to the best literature can be found online.

2 Computers in 1968 were "becoming increasingly integrated into the central activities of modern society" (Naur and Randell 1968, 3) and many at the conference were concerned that software failures would come to harm those who were now relying upon computers.

The term *Software Engineering* originates from the conference's title, which is a provocative attempt to "imply the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering" (Naur and Randell 1968, 13). In this respect, the discipline of software engineering arises as a direct response to the software crisis; an attempt to overcome the crisis through a reasoned understanding of software manufacturing.

Boehm's Curve

Barry Boehm did not attend the 1968 NATO conference but it clearly influenced him. As the attendees of the conference had done, Boehm warned in 1972 that software "was often late and unreliable, and that the costs were rising" (Whitaker 1993, 300). This was considered a "shocking conclusion at the time" (Whitaker 1993, 300) and the United States Air Force, who had commissioned the study, refused to publish the findings, which they "rejected out of hand" (1993, 300). Boehm returned four years later with a paper bearing the same title as the NATO conference: *Software Engineering* (Boehm 1976). In this paper Boehm (1976, 1126-27) once again produced graphs showing that software was becoming more expensive than the hardware it ran on. However, the paper is perhaps better known for another graph it contains, a graph that has come to be known as Boehm's curve (fig. 11; 1976, 1228; 1981, 40).

Boehm's curve (fig. 11) observes that as a computer program becomes more developed, it also becomes more difficult to change. This was the same observation Paulson (1976, 588) had made about architecture projects that same year (see chap. 2.2).[3] Paulson and Boehm's curves have the same axes, the same shape, and the same conclusion. The major difference is that Boehm's curve has supporting data while Paulson's curve is more a diagram of what he thought was happening. The data in Boehm's curve forecasts that making a change late in a software project costs one hundred times more than making the same change at the project's inception. In effect, a software project – like an architecture project – becomes substantially less flexible over time and, as a result, the programmer's capacity to make changes is greatly diminished by the increasing cost of change.

3 I can find no evidence that Paulson or Boehm knew of each other's work.

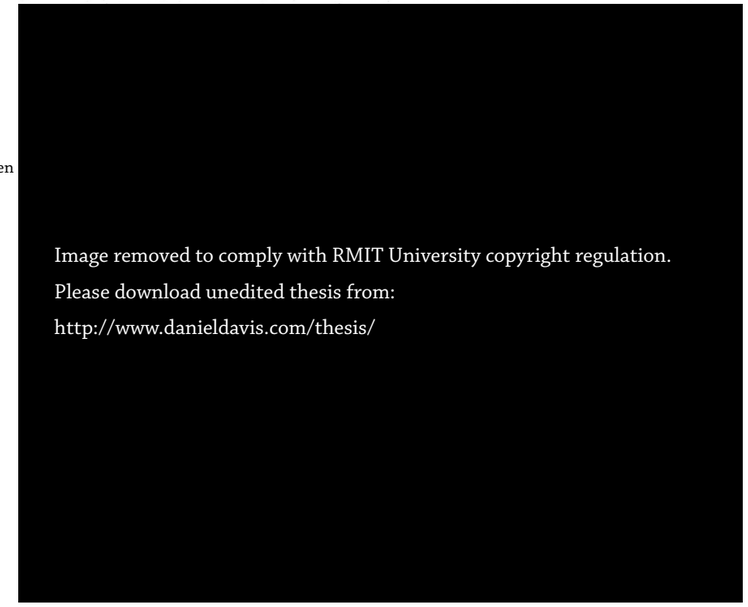


Figure 11: Boehm's curve (1981, 40). An elaboration of Boehm's earlier curve (1976, 1228). Note that Boehm plotted the data logarithmically. When plotted on a linear scale it resembles figure 12, which closely matches Paulson (fig. 9) and MacLeamy's curve (fig. 10).

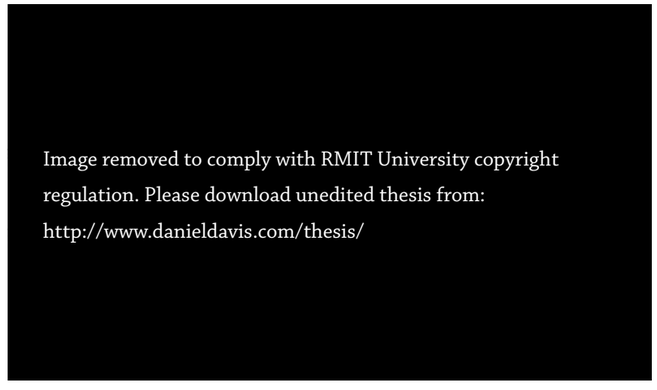


Figure 12: Boehm's curve plotted on a linear scale (Beck 1999, 26).

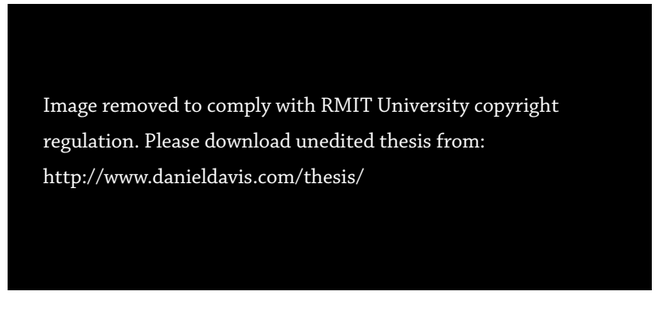


Figure 13: Beck's curve (1999, 28). There are no project stage demarcations on the horizontal axis because the relatively constant cost of change allows the project to cycle rapidly through iterations, which enables traditionally early stage activities, like developing the project requirements, to continue late into the project – and vice versa (Beck 1999, 28).

Beck's Curve

Some programmers reacted to Boehm's curve by trying to avoid change, their rationale being that if a change costs one hundred times more to make at the end of the project, then it makes sense to spend considerable time upfront preventing any late-stage changes. This is the same premise and conclusion that led MacLeamy to advocate the front-loading of architecture projects to avoid late-stage changes (see chap. 2.2). For software engineers, a common way to suppress change is with Winston Royce's (1970) waterfall method. In the waterfall method, a project is broken down into a series of stages: requirements, design, implementation, verification, and maintenance. The breakdown resembles the stages routinely used in architecture and engineering projects. Each stage is completed before proceeding to the next, with the hope being that if the requirements are finalised before commencing the design (or any other subsequent stage), then there will be no late changes from unexpected alterations to the requirements (or any other proceeding stage). Of course, finalising the requirements without seeing the design is a tricky proposition (Microsoft 2005).

Royce (1970) was aware of the waterfall method's shortcomings having originally introduced it as an example of how not to organise a software project. The waterfall method was, in fact, Royce's antithesis. Royce (1970, 329) warned that the waterfall method was "risky and invites failure", yet to his dismay, many of Royce's readers disagreed with him and instead sided with the logic of what he was arguing against. The waterfall method became what Boehm (1988, 63) describes as "the basis for most software acquisition standards," perhaps due to its clean hierarchical divisions of labour and affinity for fitting in a Gantt chart.

The method Royce (1970, 329-38) intended to advocate took the waterfall's sequential progression and broke it with eddies of feedback between the stages. This idea was extended by Boehm (1981, 41) who argued the cost of making late-stage changes was so high that in some cases it might be more effective to make successive prototypes with feedback between each iteration. Boehm (1988) later formalised this method into the *Spiral Model* of software development, which, much like Schön's Reflective Practice (1983), coils through stages of creating prototypes, evaluating the prototypes, reflecting upon the prototypes, and planning the next

stage of work. This designerly way of approaching programming forms the basis of the *Manifesto for Agile Software Development* (Beck et al. 2001a). The manifesto's fourth and final demand urges programmers to "respond to change over following a plan" (Beck et al. 2001a) – a demand that at once attacks the perceived rigidity of the waterfall method's front-loading whilst also suggesting that Boehm's cost of change curve need not be a barrier to making change. A number of programming methodologies fall under the banner of agile development, which includes Extreme Programming, Agile Unified Process, and Scrum. Kent Beck, the first signatory to the agile manifesto and the originator of Extreme Programming, captures the motivations of these methods in a book subtitled *Embrace Change*:

> The software development community has spent enormous resources in recent decades trying to reduce the cost of change—better languages, better database technology, better programming practices, better environments and tools, new notations. What would we do if all that investment paid off? What if all that work on languages and databases and whatnot actually got somewhere? What if the cost of change didn't rise exponentially over time [figure 12], but rose much more slowly, eventually reaching an asymptote? What if tomorrow's software engineering professor draws [figure 13] on the board?
> *Beck 1999, 27*

Beck provocatively suggests that Boehm's curve (fig. 12) is no longer relevant when programmers have knowledge of "better languages, better database technology, better programming practices, better environments and tools, new notations" (Beck 1999, 27). In effect, Beck says that programmers can flatten the cost of change with the body of knowledge associated with software engineering. This flattening is now known as Beck's curve (fig. 13). An important implication of Beck's curve is that the demarcations between project stages (such as: requirements, design, and production) have less importance since a relatively constant cost of change allows "big decisions [to be made] as late in the process as possible, to defer the cost of making the decisions and to have the greatest possible chance that they would be right" (Beck 1999, 28). This was a bold prediction in 1999, but increasingly studies are indicating that software engineers have gained the knowledge to lower the cost of changes. A large industry survey by the Standish Group (2012, 25) concludes "the agile process is the universal remedy for software development project failure.

Software applications developed through the agile process have three times the success rate of the traditional waterfall method and a much lower percentage of time and cost overruns." This seems to carry through into the practice of software engineering, with Dave West and Tom Grant (2010, 2) showing that programmers now use agile development more often than the waterfall method. While these results do not speak directly to Beck's curve, it is important to remember that "a flattened change cost curve makes [agile development] possible" (Beck 1999, 28). Remarkably, in only forty years, software engineering has gone from a point of crisis where the cost of late-stage changes seriously threatened the entire industry, to a point where the majority of software engineers are using a development method whose central tenet is to "welcome changing requirements, even late in development" (Beck et al. 2001b). As Beck (1999, 27) points out, the road out of the software crisis was "decades [of] trying to reduce the cost of change" now captured in an extensive body of knowledge related to software development.

3.2 The Software Engineering Body of Knowledge

There is reason to suspect the body of knowledge concerning software engineering may also apply to architects using parametric models. Frederick Brooks (2010) makes a similar connection in his book *The Design of Design*, where he recounts designing his house and relates this to his experiences managing the design of IBM's System/360 architecture (2010, 257-346). Brooks (2010, 21) says change is inevitable for both programmers and architects since they both normally begin with "a vague, incompletely specified goal, or primary objective" only clarified through iteratively creating and changing prototypes. These difficulties are compounded in the two practices, both by the fact that the cost of change generally rises exponentially as a project progresses, and by the fact that undetermined outcomes need to be expressed in logically precise instructions for the computer. While this problem is relatively new for architects creating parametric models, the same problem has challenged software engineering for decades. Evidence suggests that the knowledge software engineers have gained during this time allows them some control over the cost of change. This knowledge could potentially do the same in architecture.

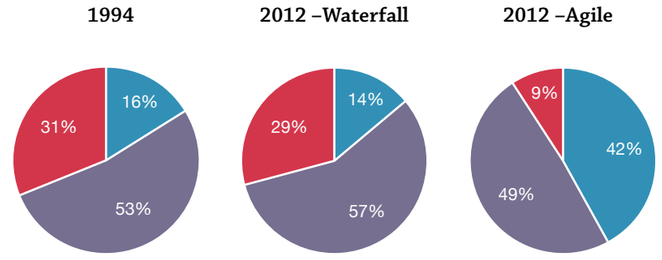


Figure 14: The success and failure rates of software projects according to The Standish Group's industry survey (1994; 2012).

Successful projects – delivered on-time, on-budget, and with the planned features.

Challenged projects – either: over time, over budget, or lacking features.

Failed projects – the project was abandoned.

There are two main caveats in applying software engineering to parametric modelling. One caveat is that software engineers are often not particularly successful at what they do. On average, 49% of software projects using an agile development process will encounter significant challenges while 9% will fail outright. Just 42% of software projects are delivered on time, on budget, and with the specified features (fig. 14). While a 42% success rate may sound low, the Standish Group (2012, 25) says this represents the "universal remedy for software development project failure" principally because software engineers have historically had a success rate of only 16% (fig. 14; The Standish Group 1994). Thus, even software engineers following the best practices still encounter trouble more than they encounter success.

Another important caveat is that creating software is similar, but not identical, to creating architecture. Broadly speaking, some common points of difference include the following:

- **The user:** Software engineers tend to make software used by other people, whereas architects generally create parametric models for either themselves or for their colleagues.
- **The product:** Software engineers make software but architects ultimately make architecture rather than parametric models. While software may be evaluated in and of itself, a parametric model is typically valued for the architecture it produces.
- **Team size:** Software engineering teams range from lone individuals building an app, to thousands of developers creating an operating system. In comparison, parametric models are generally made by teams at the smaller end of this range.
- **Project lifetime:** Software engineering projects may last anywhere from a few minutes to a few decades, whereas the code in a parametric model is unlikely to persist beyond a few years (or perhaps even months).

There will be numerous exceptions to these broad generalisations. The point, however, is that while architects and software engineers share similar challenges, not all of software engineering is equally relevant to the idiosyncratic circumstances of parametric modelling. In this section I outline the software engineering body of knowledge and hypothesise about which parts are most pertinent to the practice of parametric modelling.

Classifying Knowledge

There have been a number of attempts to classify knowledge relating to software engineering. In 1997, the Institute of Electrical and Electronic Engineers (IEEE) formed a committee tasked with creating the first "comprehensive body of knowledge for software engineering" (Hilburn et al. 1999, 2). This was a controversial undertaking. The Association for Computer Machinery (ACM) feared the body of knowledge "would likely provide the basis for an exam for licensing software engineers as professional engineers" (ACM 2000). The ACM, like many others, withdrew their support of the project. The IEEE committee's four-year schedule dragged into seven years of deliberation. Meanwhile, Thomas Hilburn et al. (1999) sidestepped the IEEE committee to produce their own, and the first, *Software Engineering Body of Knowledge Version 1.0* (SWEBOK.1999; fig. 15). This document captured the expected knowledge of a programmer who has spent three years in the industry, and was released in conjunction with Donald Bagert et al. (1999) corresponding *Guidelines for Software Engineering Education Version 1.0* (SE.1999). Eventually, in 2004, a similar pair of documents was published by the IEEE committee: Alain Abran and James Moore's (2004) *Guide to the Software Engineering Body of Knowledge* (SWEBOK.2004) along with Jorge Díaz-Herrera and Thomas Hilburn's (2004) *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (SE.2004).

The relationship between the various SWEBOK is shown in figure 15. While the taxonomies are different, they all use the waterfall method as a template for classifying the software engineering process. This is not an endorsement of the waterfall method since the division of knowledge need not prescribe its deployment. For example, projects using an agile methodology necessarily apply knowledge of planning and coding and testing, although not in the same linear fashion as projects using the

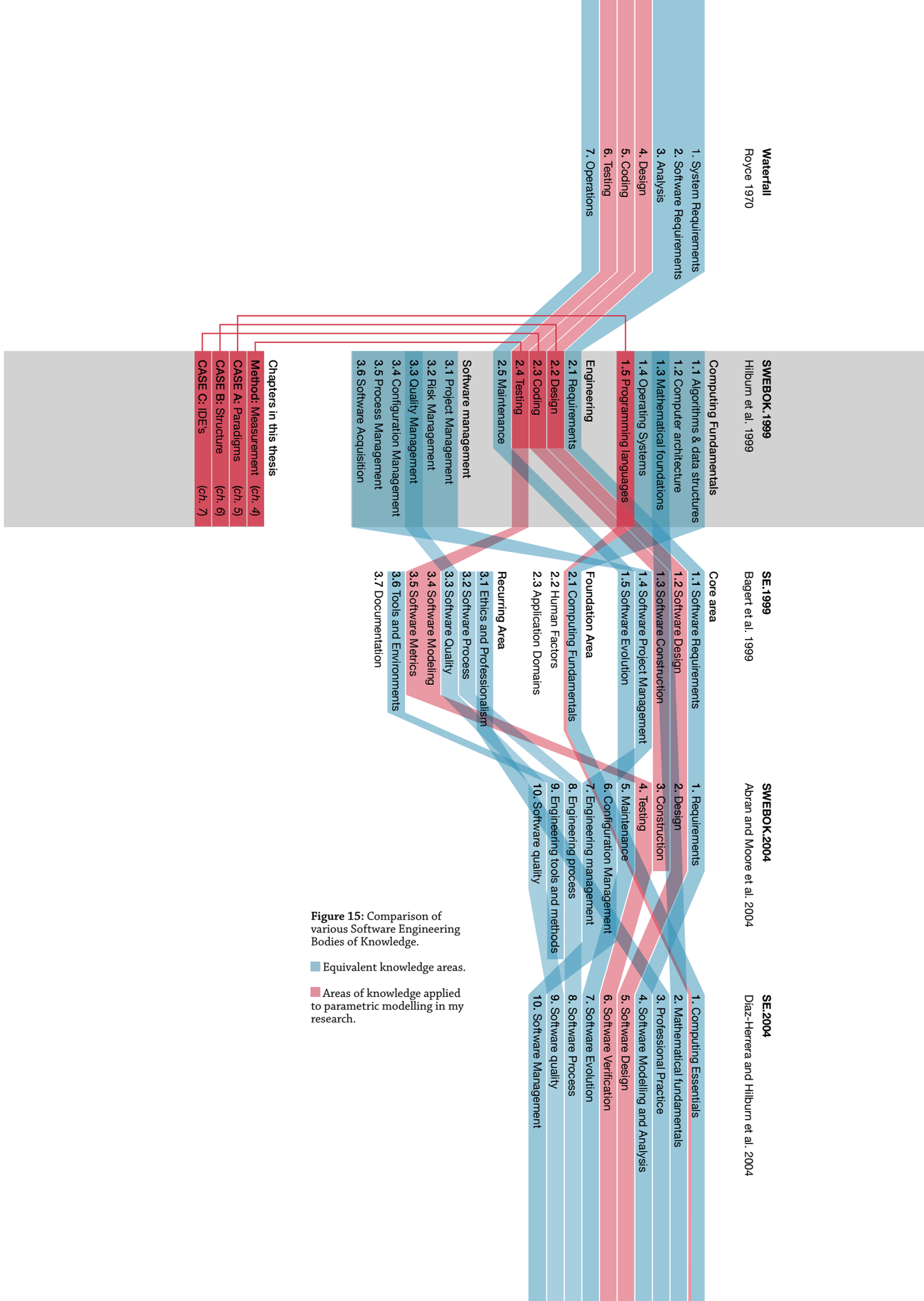


Figure 15: Comparison of various Software Engineering Bodies of Knowledge.

Equivalent knowledge areas.

Areas of knowledge applied to parametric modelling in my research.

waterfall method. With each SWEBOK agnostically employing the waterfall method's stages, the key differences between the various SWEBOK lie in the classification of knowledge not pertaining to the waterfall's stages. The SWEBOK.1999 clearly segregates these areas, with waterfall's stages confined to the engineering category, which is separated from the computing fundamentals category and the software management category. While software management reappears in all the other SWEBOK, the computing fundamentals category is unique to the SWEBOK.1999 and covers areas of knowledge – like computer hardware and programming languages – that are potentially applicable to parametric modelling. For this reason, I have selected the SWEBOK.1999 to use in the following pages as I hypothesise about which parts are also applicable to architects creating parametric models. However, given the relative homogeneity of the various SWEBOK, I would expect similar results from using any of the other SWEBOK.

SWEBOK.1999
Hilburn et al. 1999

Computing Fundamentals
1.1 Algorithms & Data Strct.
1.2 Computer Architecture
1.3 Mathematical Fndn.
1.4 Operating System
1.5 Programming Languages

Engineering
2.1 Requirements
2.2 Design
2.3 Coding
2.4 Testing
2.5 Maintenance

Software management
3.1 Project Management
3.2 Risk Management
3.3 Quality Management
3.4 Configuration Mgmt.
3.5 Process Management
3.6 Software Acquisition

1. Computing Fundamentals

The *Computing Fundamentals* [1] category of the SWEBOK.1999 covers the foundational theories and concepts of software engineering. Many parts of this category are so essential to computing that they already necessarily contribute to parametric modelling. For instance, *Computer Architecture* [1.2] concerns the underlying structure of a computer, which includes the way transistors are laid out to allow more intensive calculations, and how networks exchange data to permit remote collaboration. Deriving the benefits of this knowledge requires no intervention from the software engineer or parametric modeller since it is encapsulated within a computer's hardware. The same is true of both the *Mathematical Foundation* [1.3], which provides the formal logic to programming, and of *Operating Systems* [1.4], which provides the framework supporting the software. While the *Computer Architecture* [1.2], the *Mathematical Foundations* [1.3], and *Operating Systems* [1.4] have made large contributions to software engineering, these contributions come – in many ways – independent of the actions from software engineers. By proxy, designers are already benefiting from these areas of *Computing Fundamentals* [1] whenever they purchase new computer hardware or invest in new operating systems.

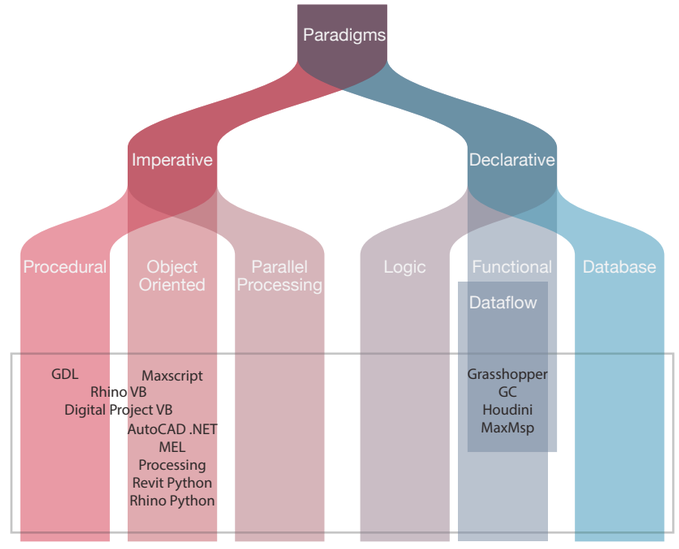


Figure 16: The programming languages architects use categorised by Appleby and VandeKopple's (1997, xiv) taxonomy of programming paradigms.

Algorithms and Data Structures [1.1] are not built into hardware and must instead be actively fashioned for a particular task. Considerable research has gone into tailoring *Algorithms* [1.1.2] and *Data Structures* [1.1.1] for parametric modelling. Examples of existing *Algorithms* [1.1.2] used in parametric modelling include algorithms for propagating changes through parametric models (Woodbury 2010, 15-16), rationalisation algorithms for simplifying complex surfaces (Wallner and Pottmann 2011), algorithms for simulating physical properties (such as: Piker 2011), and many proprietary algorithms buried in commercial software and geometry kernels (such as: Aish et al. 2012). Similar work has been done on *Data Structures* [1.1.1] to develop specialised file formats for things like sharing BIM models, describing B-rep geometry, and saving parametric models. While there is scope to further these existing *Algorithms* [1.1.2] and *Data Structures* [1.1.1], any improvements are likely to be refinements of what already exists. Given the maturity of the research in this area, I see few opportunities to address the flexibility of parametric models through making further contributions to *Algorithms and Data Structures* [1.1].

As with *Algorithms and Data Structures* [1.1], there are already many *Programming Languages* [1.5] for architects creating parametric models. Every programming language has a unique style for expressing concepts, which is called the language's *Programming Paradigm* [1.5.2] (fig. 16). The paradigm influences how problems are solved in a particular language.

For instance, Appleby and VandeKopple (1997, 7) show how the United States Department of Defense addressed problems of “unmaintainable” and “fragile” software by creating a new multi-paradigm programming language, Ada (first released 1980). Appleby and VandeKopple (1997, xiv) divide programming paradigms – as many others do – into imperative paradigms and declarative paradigms (fig. 16). I will explain these denominations later in chapter 5 but for now it suffices to say that there is a broad taxonomy of possible programming paradigms. Currently architects only have access to two narrow bands of programming paradigms (see distribution in figure 16): the major textual CAD programming languages[4] are all predominantly imperative with a bias towards procedural programming; whereas, the major visual CAD programming languages[5] all reside in a very narrow subsection of declarative programming known as dataflow programming. While the two bands of paradigms occupied by CAD programming languages are well researched, they are ultimately limited. For architects this means they have a confined range of styles available to express ideas programmatically. This presents an opportunity to expand the practice of parametric modelling by borrowing new programming paradigms from software engineers.

SWEBOK.1999
Hilburn et al. 1999

Computing Fundamentals
1.1 Algorithms & Data Strct.
1.2 Computer Architecture
1.3 Mathematical Fndn.
1.4 Operating System
1.5 Programming Languages

Engineering
2.1 Requirements
2.2 Design
2.3 Coding
2.4 Testing
2.5 Maintenance

Software management
3.1 Project Management
3.2 Risk Management
3.3 Quality Management
3.4 Configuration Mgmt.
3.5 Process Management
3.6 Software Acquisition

2. Software Product Engineering

The *Software Product Engineering* [2] category of the SWEBOK.1999 describes the activities involved in producing software. These activities are categorised by the phases of the waterfall method. As I explained in the preceding pages, the divisions do not prescribe that software engineers use the waterfall method since these categories are intended to capture the knowledge necessary for producing software independent of the overall programming process.

Software Product Engineering’s [2] first area of knowledge is *Software Requirements Engineering* [2.1], which pertains to the creation of project briefs. By and large there is nothing particularly remarkable about the way programmers create briefs. Like in other disciplines, they analyse the situation [2.1.1], generate requirements [2.1.2], and write specifications

4 This includes: 3dsMax: Maxscript; Archicad: GDL; Autocad: AutoLISP; Digital Project: Visual Basic; Maya: Maya Embedded Language; Processing: Java; Revit: Visual Basic & Python; Rhino: Visual Basic & Python; Sketchup: Ruby.
5 This includes: Grasshopper; GenerativeComponents; Houdini; and MaxMsp.

[2.1.3]. While these are important steps in producing software, they are a process architects are likely already adept at.

From the *Requirements* [2.1] flows the *Software Design* [2.2], which in software engineering concerns the design of interfaces as well as the structure of code, data, and algorithms. Spending time structuring code rather than writing code has not always been a pastime of programmers. Prior to the software crisis, most programming languages (like FORTRAN) did not have a syntax for describing structure. The resulting programs generally had what Bertrand Meyer (1997, 678) calls, the “unmistakable ‘spaghetti bowl’ look” of logic weaving haphazardly through unstructured code. Edsger Dijkstra (1968, 148) called the unstructured jumps “harmful” and “too much an invitation to make a mess of one’s program” (an observation he made in the same year as the NATO *Software Engineering* conference). In the ensuing years, most programming languages have adopted Böhm and Jacopini’s (1966) concept of enforcing structure with conditionals, loops, and subprograms. Meyer (1997, 40-46) argues that these structures help to decompose incomprehensibly large problems into vastly more understandable smaller structured chunks.[6] Despite these benefits, most parametric software has only rudimentary support for structure, which the vast majority of architects – like programmers prior to the software crisis – shun in favour of unstructured models (the low rates of structure are revealed and discussed in chapter 6.3). Woodbury, Aish, and Kilian’s (2007) *Some Patterns for Parametric Modeling* suggests some common structures for parametric models, however, their structures are predominately focused on solving architectural design problems rather than addressing the problems of unstructured code. Accordingly, there remains significant scope to implement relatively straightforward structuring techniques on parametric models, which (based on evidence from similar interventions during the software crisis) may improve the understandability of parametric models.

The actual act of writing computer code is covered in *Code Implementation* [2.3.1], a subsection of *Software Coding* [2.3]. At first glance, writing code may seem worthy of a more prominent place in the SWEBOK.1999, especially given that writing code is one of the defining jobs of a software

6 Meyer (1997, 40-46) cites benefits to code decomposition, composition, understandability, continuity, and protection, which I will discuss in further in chapter 6.2.

engineer. Yet, the positioning of *Code Implementation* [2.3.1] in such a minor category indicates how much ancillary knowledge goes into successfully writing code. This is an important observation when considering what architects need to learn in order to create a parametric model, and it is a point I will return to in the discussion (chap. 8.4) as I contrast the education of software engineers with the education of architects learning to use parametric models.

The *Code Implementation* [2.3.1] category also encompasses tools programmers use to write code. These tools, known as Integrated Development Environments (IDE), assist programmers by managing the compiling and debugging of code, as well as providing feedback to aid code comprehension (such as: pointing out possible coding errors, or explaining the meaning of a particular programming command). In contrast, Leitão, Santos, and Lopes (2012, 143) say "the absence of a (good) IDE" for parametric modelling "requires users to either remember the functionality or read extensive documentation." They go on to say, "an iterative write-compile-execute cycle," implemented in most parametric modelling environments, "results in non-interactive development" (2012, 143). These limitations in the tools architects use to create parametric models could be addressed by borrowing concepts like live-debugging, live-programming, and other innovations from the IDEs software engineers use.

Software Coding [2.3] has two additional sections: *Reuse* [2.3.2], and *Standards and Documentation* [2.3.3]. Both of these sections are related to *Software Design* [2.2]. *Reuse* [2.3.2] relates to how the program has been structured and particularly whether modules of code can be extracted and shared. The structure also plays a role in *Standards and Documentation* [2.3.3] since these are tied to the levels of abstraction in the structure. Both *Reuse* [2.3.2] and *Standards and Documentation* [2.3.3] help reinforce the importance of well-structured programs and give more impetus to investigate the structure of parametric models.

Software Testing [2.4] involves verifying code correctness. Programmers like to automate this process, either by using metrics for measuring performance [2.4.4], or by automated unit testing of the code itself [2.4.1, 2.4.2], or even with quantitative experiments like A/B testing user behaviour. Anecdotally, architects seem to test their models by manually verifying the outputs, which can lead to problems like change blindness

SWEBOK.1999
Hilburn et al. 1999

Computing Fundamentals
1.1 Algorithms & Data Strct.
1.2 Computer Architecture
1.3 Mathematical Fndn.
1.4 Operating System
1.5 Programming Languages

Engineering
2.1 Requirements
2.2 Design
2.3 Coding
2.4 Testing
2.5 Maintenance

Software management
3.1 Project Management
3.2 Risk Management
3.3 Quality Management
3.4 Configuration Mgmt.
3.5 Process Management
3.6 Software Acquisition

(see chap. 2.3). Schultz, Amor, and Guesgen (2009, 402) demonstrate that testing methods "inspired by research in software engineering" may be applied to "qualitative spatial" problems. While there is considerable opportunity for further research in this area, given my focus on parametric model flexibility, I have elected to look only at *Software Testing* [2.4] in relation to measuring model flexibly with software metrics [2.4.4] (see chap. 4).

The final category in *Software Product Engineering* [2] is *Software Operations and Maintenance* [2.5], which embodies "concepts, methods, processes, and techniques that support the ability of a software system to change, evolve, and survive" (Hilburn et al. 1999, 25). In a similar manner, my research focuses on the change, evolution, and survival of both software and parametric models. In software engineering, the *Software Maintenance Process* [2.5.3] employs a "process [that] would include phases similar to those in a process for developing a new software product" (Hilburn et al. 1999). Thus, while *Software Operations and Maintenance* [2.5] is a distinct stage of *Software Product Engineering* [2], and a stage that closely resembles the goals of my research, the actual knowledge of operations and maintenance is already deployed in the prior stages of *Software Product Engineering* [2].

3. Software Management

Software Management [3] is the last major category of the SWEBOK.1999. Many of the same management challenges reoccur in software engineering and parametric modelling. These include more general challenges, such as managing a creative process whilst adhering to a budget, a schedule, and guarantees of quality; and these also include more specific challenges, like managing the development of code when the programming language requires precision but the outcome is uncertain. Accordingly, the management strategies employed by software engineers often have rough equivalence to strategies employed by architects. For example, the waterfall method has similar stages and a similar shift in effort to MacLeamy's front-loading, and agile development has a similar pattern of iterative prototyping present in Schön's reflective practice.

However, within these general areas of agreement, there are idiosyncrasies to the specific management practices of software engineers. In *Software Quality Management* [3.3] (which overlaps with *Testing* [2.4]), software engineers emphasise automated quantitative measures of quality, either through unit testing to validate the code or through metrics to measure code quality objectively (these are applied to parametric models in chapter 4.3). And in *Software Process Management* [3.5] there is a degree of formalism around the design processes that would be unfamiliar to most architects. For instance, in the Scrum development process (a popular form of agile development) the inventors, Jeff Sutherland and Ken Schwaber (2011, 6-10), specify everything from the number of days a design cycle should last (a month), to the ideal team size (less than nine people), to the length of daily meetings (fifteen minutes). Since these management processes are so tuned to the nuances of programming, further research is required to establish whether they also translate to the nuances of parametric modelling.

Whilst *Software Management* [3] is undoubtedly a ripe area of investigation in the context of parametric modelling, it is an investigation I will leave for others to undertake. I have decided to limit my thesis primarily to the study of *Computer Fundamentals* [1] and *Software Product Engineering* [2] because understanding these technical issues is quite different to understanding the ethnographic issues of management. Covering both inside one thesis is unlikely to do justice to either. For this reason I will touch on only some of the ideas in *Software Management* [3], notably around *Software Quality Management* [3.3], but it will not be a primary focus for the remainder of this thesis.

SWEBOK.1999
Hilburn et al. 1999

Computing Fundamentals
1.1 Algorithms & Data Strct.
1.2 Computer Architecture
1.3 Mathematical Fndn.
1.4 Operating System
1.5 Programming Languages

Engineering
2.1 Requirements
2.2 Design
2.3 Coding
2.4 Testing
2.5 Maintenance

Software management
3.1 Project Management
3.2 Risk Management
3.3 Quality Management
3.4 Configuration Mgmt.
3.5 Process Management
3.6 Software Acquisition

3.3 Conclusion

The software crisis recalls many of the same challenges of parametric modelling. For software engineers, the improvements in computation during the 1960s resulted in more software being developed. The software was generally growing larger, being written in more abstracted languages, and running on-top better hardware. However, rather than programming becoming easier, these improvements intensified the difficulty of simply writing software (Wirth 2008, 33). Like architects working with parametric models, software engineers struggled to make changes within the logical rigidity of programming. These difficulties were amplified by the cost of change rising exponentially during a project – a phenomena highlighted by Boehm (1976; fig. 11) in a graph that resembles similar graphs by Paulson (1976; fig. 9) and MacLeamy (2001; fig. 10).

The software crisis gave rise to software engineering, a discipline dedicated to understanding the manufacture of software (Naur and Randell 1968, 13). Since demarcating this area of knowledge in the 1960s, software engineers have steadily become more successful at producing software (fig. 14; The Standish Group 1994 & 2012). Software engineers now postulate that that they can lower the cost of change to the point where the vertical asymptote of Boehm's curve bends horizontal (fig. 13; Beck 1999, 27). Such radical transformations in software engineering arise from knowledge gained during decades of work studying the software engineering process.

The knowledge that has transformed software engineering is classified in the *Software Engineering Body of Knowledge Version 1.0* (Hilburn et al. 1999). Somewhat surprisingly, the act of writing code occupies a very small sub-section [2.3.1] of this classification; a position that underscores the breadth of knowledge (besides simply knowing how to program) required for successfully developing software. Some areas of knowledge, like *Software Management* [3], have direct correlations to the design process. Other areas, like certain aspects of *Computing Fundamentals* [1], are so essential to anything involving a computer that architects already necessarily benefit from them. However, large portions of the SWEBOK.1999 are largely without precedent in the practice of parametric modelling. In this chapter I have identified a number of knowledge areas that are potentially applicable

to parametric modelling while being accessible within the technical and temporal constraints of a PhD thesis. These are:

1.5 Programming Languages
2.2 Software Design
2.3 Software Coding
2.4 Testing

Programming Languages [1.5], *Software Design* [2.2], and *Software Coding* [2.3] are the respective focus of the three case studies in chapters 5, 6, & 7. Specifically, chapter 5 explores the impact of under-utilised *Programming Paradigms* [1.5.2], chapter 6 considers how the structure of *Software Design* [2.2] may apply to a parametric model, and chapter 7 investigates how *Code Implementation* [2.3.1] environments inform the development of parametric models. Each of these chapters aims to assess the influence the respective area of knowledge has on the flexibility of various parametric models. In order to measure flexibility, I draw upon the knowledge area of *Testing* [2.4], the focus of the following chapter.

SWEBOK.1999
Hilburn et al. 1999

Computing Fundamentals
1.1 Algorithms & Data Strct.
1.2 Computer Architecture
1.3 Mathematical Fndn.
1.4 Operating System
1.5 Programming Languages

Engineering
2.1 Requirements
2.2 Design
2.3 Coding
2.4 Testing
2.5 Maintenance

Software management
3.1 Project Management
3.2 Risk Management
3.3 Quality Management
3.4 Configuration Mgmt.
3.5 Process Management
3.6 Software Acquisition

4 Measuring Flexibility

Measuring a parametric model's flexibility is a somewhat challenging proposition. There is no agreed upon definition of flexibility, nor is there any existing way to measure it. Furthermore, as I outlined in the introduction (chap. 1), flexibility is often intwined with the circumstances of a project, making it hard to clearly observe what is happening. Flexibility remains largely enigmatic.

In this chapter I outline a framework for observing the flexibility of parametric models. I begin by proposing a research method that relies upon triangulation between case studies to mitigate some of the circumstantial challenges of observing flexibility. In the second half of the chapter I draw upon concepts encapsulated in the *Testing* [2.4] section of the SWEBOK.1999. Borrowing from software engineering, I outline a suite of quantitative and qualitative research instruments for measuring various types of flexibility in a parametric model. In aggregate, the research method and research instruments will serve as a foundation for observing flexibility in the case studies presented during chapters 5, 6, & 7.

4.1 Research Method

The flexibility of a parametric model can be hard to observe. To date, the best observations have come from architects working on projects where the model became inflexible and failed. While architects can be reluctant to talk about their failures, the few who have done so (discussed in chapter 2.3) prove useful in identifying the challenges associated with parametric modelling. However, in coming to understand why parametric models are failing, these reports tend to offer little insight beyond documenting general symptoms – a major change breaks the model, the model is hard to share, there is a need to anticipate changes whilst parametric modelling (see chap. 2.3). Most of these observations come in the course of other research; the authors had not set out to study model flexibility and while they were able to identify the symptoms of inflexibility, they generally

lacked the controls necessary to isolate the contributing factors. Herein lies the paradox: flexibility is intertwined with the design process yet the circumstances of the design process make it difficult to obtain confident observations of parametric model flexibility.

In the introduction (chap. 1) I highlighted that software engineers often conduct similar studies to my own. When doing so, they face an analogous challenge of trying to understand the intricate interrelationships between people, code, and computers. To make sense of these relationships, Tim Menzies and Forrest Shull (2010, 3) say that software engineers often seek elegant, repeatable, statistical studies (even the name software *engineering* has connotations of this positivist perspective). While such an approach works for certain aspects of software engineering (like Boehm's [1976] empirical analysis regarding the cost of change [fig. 11]) Edsger Dijkstra (1970, 1) argues that for studies related to practice, it is problematic to study small, idealised problems and then generalise them by concluding with the assumption: "… and when faced with a program a thousand times as large, you compose it in the same way." Dijkstra (1970, 2) contends that the "widespread underestimation" of project-based circumstances, in research prior to 1970, was "one of the major underlying causes of the current software failure [the software crisis]." Therefore, as I argued in the introduction (chap. 1), attempting to create a simplified, controlled, and isolated study may eliminate the best opportunities to observe how parametric flexibility manifests in practice.

In the introduction I posited that case studies might offer a way to understand flexibility without needing to isolate research from practice. While this may be closer to social science than the *hard science* origins of software engineering, Andrew Ko (2010, 60) argues such an approach is "useful in any setting where you don't know the entire universe of possible answers to a question. And in software engineering, when is that *not* the case?" A salient example from software engineering is Frederick Brooks's (1975) *The Mythical Man-Month : Essays on Software Engineering* where Brooks reflects upon his experiences managing IBM's System/360. These reflections, in a similar spirit to Schön's (1983) notion of *reflection on action*, provide other researchers and practitioners with an insight into managing a large software project that would be unobtainable from just

examining specific parts in isolation. Such a method has a constructivist worldview where, according to Creswell and Clark (2007, 24), multiple observations taken from multiple perspectives build inductively towards "patterns, theories, and generalizations."

A key component of case study research is selecting a suite of cases that ensure the validity of anything built inductively on-top of them. Given the spectrum of issues concerning parametric models, a single case study – or even a collection of case studies – is unlikely to be entirely representative. In research projects where instrumental case studies cannot be found, Robert Stake (2005, 451) encourages researchers to select case studies that "offer the opportunity to learn" because "sometimes it is better to learn a lot from an atypical case than a little from a seemingly typical case." In my research I want to learn about applying software engineering knowledge to the practice of parametric modelling. In the previous chapter (chap. 3) I hypothesised about which aspects of the software engineering body of knowledge are most likely to influence a parametric model's flexibility. In selecting the projects to test this knowledge, the best opportunity to learn about flexibility is seemingly presented by projects likely to encounter difficulties. According to the factors I identified in chapter 2.3, the projects most fated for trouble are those where the following are applicable: the outcomes cannot be anticipated from the start, major changes are likely, the model is large or complicated, change blindness occurs, and the model is shared. A final criterion for selecting the cases is that the projects have to be accessible and manageable within the three-year period of my PhD candidature. With these criteria in mind I have selected the following three case studies:

- **Case A:** *Realignment of the Sagrada Família frontons*
  A project that involves developing a relatively complicated parametric model to refine an existing model of the Sagrada Família's frontons. The project has strict tolerances but there is also ambiguity as to what the realignment will involve, which causes uncertainty regarding changes to the project. On this project I investigate how *Programming Paradigms* [1.5.2] impact the construction and modification of parametric models. See chapter 5.

- **Case B:** *The Dermoid pavilion*
  The Dermoid pavilion is a collaborative design project involving over a dozen researchers from Melbourne and Copenhagen. The pavilion's wooden reciprocal frame is formidably hard to model. Furthermore, the models need to remain flexible enough to accommodate major changes from a range of authors over a period of a year. On this project I explore how *Software Design* [2.2] influences the understandability of parametric models that are used in collaborative environments. See chapter 6.

- **Case C:** *The hyperboloid sound wall.*
  Change blindness during the design of the hyperboloid sound wall led to significant problems during the wall's construction. I revisit this project and consider how *Code Implementation* [2.3.1] may help improve the interactivity of parametric modelling. See chapter 7.

While the three case studies are not necessarily representative of how parametric models are typically employed in architecture projects, the slightly atypical nature of the three case studies means that they touch on many of the major issues concerning parametric modelling. In aggregate, these cases make up what Robert Stake (2005, 446) calls a "collective case study" where multiple projects "are chosen because it is believed that understanding them will lead to better understanding, and perhaps better theorising, about a still larger collection of cases." My intention in selecting the case studies has been to choose three situations where key challenges of parametric modelling are likely to be exhibited because I believe understanding the relationship between parametric modelling and software engineering in these challenging circumstances may lead to a better understanding of this relationship more generally.

4.2 Research Instruments

A research instrument, as defined by David Evan and Paul Gruba (2002, 85), is any technique a "scientist might use to carry out their 'own work'." Typical examples include interviews, observations, and surveys. My 'own work' is to understand how knowledge taken from software engineering impacts the flexibility of the parametric models from the various case studies. To help me carry out this work, ideally there would be a research instrument for measuring parametric flexibility. Unfortunately, none exist.

Flexibility concerns, at its essence, the ease with which a model can change. In a book titled *Flexible, Reliable Software*, Henrik Christensen (2010, 31) argues that all models can be changed since "any software system can be modified (in the extreme case by throwing all the code away and writing new software from scratch)." These extreme cases are fairly easy to identify: they are the moments when the designer has no other option but to rebuild the model (such as the examples discussed in chapter 2.3). Yet, there is a nuanced spectrum of flexibility leading up to this extreme. Christensen (2010, 31) says that while any model can be changed "the question is at what cost?" This is a question Boehm, Paulson, and MacLeamy all asked when they created their cost of change curves. Ostensibly, the cost of a modification may seem synonymous with the time taken to make a modification – if a model facilitates faster changes then presumably these changes cost less and the model is therefore more flexible than the alternatives. But the time taken to make a change is only one component of any particular modification's cost. If a change results in a model that is more complicated, less flexible, and more difficult to share, the long-term cost may be significantly higher than simply the time the change took to make. Software engineers call the combination of factors: *code quality*. In the following pages I outline some of the key quantitative and qualitative research instruments for measuring code quality. Collectively these instruments help triangulate an understanding of flexibility that goes beyond simply measuring how long it takes to make a change.

4.3 Quantitative Flexibility

In an attempt to understand software quality, software engineers have invented numerous quantitative methods for evaluating various aspects of their code. There are at least twenty-three unique measures of software quality categorised in Lincke and Welf's (2007) *Compendium of Software Quality Standards and Metrics*, and over one hundred in the ISO/IEC 9126 standard for *Software Product Quality* (ISO 2000). While many of these metrics are only applicable in specific circumstances,[1] a few are used almost universally by software engineers. In the following paragraphs I take six of the key quantitative metrics and explain how they apply to parametric modelling.

Construction Time

Construction time measures the time taken to build a model from scratch. Clearly there are benefits to a shorter construction time, particularly if a model gets rebuilt frequently during a project. Different users are likely to have different construction times since the user's familiarity with a modelling environment helps determine how quickly they can build a model. In general, the construction time for a parametric model is often longer than with other modelling methods because the process of creating parameters and defining explicit functions typically requires some degree of front-loading (see chap. 2.3), which is often recouped through shorter modification times.

Modification Time

The modification time measures the time taken to change the model's outputs from one instance to another. Shorter modification times allow designers to make changes more quickly, which is one of the principle reasons for using a parametric model. Changes may involve modifying the values of the model's parameters and they may involve the generally more arduous process of modifying the model's explicit functions. When designers talk about trying to 'anticipate flexibility' (see chap. 2.3) they are normally talking about reducing the subsequent modification time by arranging the model so that changes occur through manipulations of the parameters rather than the often slower manipulations of the functions. An important point here is that modification time is highly dependent upon the model's organisation, and particularly how this is impacted by the vestigial buildup of changes. Furthermore, as with construction time, the user's familiarity with a model and modelling environment has a great bearing on the modification time.

1 The ISO/IEC 9126 standard has metrics for everything from how easy the help system is to use, to how long the user waits while the code accesses an external device.

Latency

Latency is the period of time the users waits – after making a change – to see the model's latest output. The latency is caused by the computer performing the calculations necessary to generate the model's output. Often these calculations result in an imperceptible latency, but on computationally intensive models the latency can last minutes and even hours. Latency is important because designers sometimes fail to observe changes to a model, particularly if there is a pause between making a change and the change becoming visible (see chap. 2.3; Nasirova et al. 2011; Erhan, Woodbury, and Salmasi 2009). For a model to feel interactive, research suggests that the latency should ideally be less than a tenth of a second and certainly not much more than one second (Miller 1968, 271; Card, Robertson, and Mackinlay 1991, 185). In many cases this is impossible given the computational demands of various geometric calculations, the limitations of computer hardware, and the bottlenecks in the underlying algorithms of parametric modelling environments.

Dimensionality

Dimensionality is a tally of a model's parameters. Or, put another way, the number of dimensions in the model's search space. In chapter 2.3 I explained how a designer has to balance a model's dimensionality since, on one hand, parameters can help improve modification times (a higher dimensionality is better) and yet, on the other hand, too many parameters makes finding and modifying any individual parameter unwieldy (a lower dimensionality is better). Therefore, an ideal parametric model would encompass all the variations the designer wants to explore within the smallest dimensionality possible.

Size

Software engineers commonly measure a program's size by counting the lines of code (LOC). It is a somewhat imprecise measure because programs can be rewritten to occupy more or fewer lines of code. This led Steven McConnell (2006, 150) to argue, "the LOC measure is a terrible way to measure software size, except all other ways to measure size are worse." While there is a degree of imprecision, the LOC measurement is a frequently used instrument for quickly understanding the relative size of software. Ordinarily, a smaller LOC is better since the LOC measurement correlates highly with both code complexity (van der Meulen and Revilla 2007) and the number of coding errors (El Emam et al. 2001) – in essence, more lines of code provide more opportunities for things to go wrong.

In my research I use the physical lines of code measure – the number of lines of code actually written in the programming environment. In visual programming languages a node can be considered roughly equivalent to a line of code. Thus, I measure the size of visual programs throughout my research by counting the number of nodes. This allows comparisons between various visual programs however, given the differences between textual lines of code and visual nodes, comparisons cannot be made between the sizes of textual and visual programs.

Cyclomatic Complexity

Cyclomatic complexity is a core software engineering metric for measuring code structure. In technical terms, the cyclomatic complexity is the number of independent paths through a directed acyclic graph (DAG). This can be seen visually in figure 17 & 18. The cyclomatic complexity is typically calculated using Thomas McCabe's (1976, 314) formula:

$$\mathrm{CC}(G) = e - n + 2p$$

Where:

- G: the graph.
- e: number of edges. I count parallel edges between identical nodes (duplicate edges) as a single edge.
- n: number of nodes. I do not count non-functional nodes such as comments in text boxes.
- p: number of independent graphs (parts).

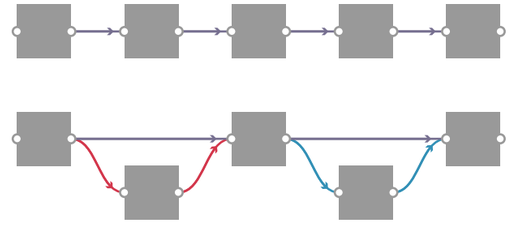


Figure 17: A directed acyclic graph comprised of a single path, which gives it a cyclomatic complexity of one.

Figure 18: A graph with the same number of nodes as in figure 17 but with three distinct paths (each colour coded). This graph therefore has a cyclomatic complexity of three.

McCabe's formula assumes the DAG has only one input node and one output node, which is infrequently the case with parametric models. In an appraisal of common modifications to McCabe's original formula, Henderson-Seller and Tegarden (1994, 263) show that "additional (fictitious) edges" can be introduced to deal with multiple inputs and outputs. Thus the cyclomatic complexity formula becomes:

$$\mathrm{CC}(G) = (e + (i\text{-}1) + (u\text{-}1)) - n + 2$$

Which (assuming p to be 1) simplifies to:

$$\mathrm{CC}(G) = e + i + u - n$$

Where:

- i: number of inputs (dimensionality).
- u: number of outputs.

The cyclomatic complexity indicates how much work is involved in understanding a piece of code. For instance, the DAG in figure 17 can be understood by reading sequentially along the single path of nodes. But understanding the more complicated DAG in figure 18 requires simultaneously reading through three different paths as they diverge and converge back together. While it may be possible to comprehend how three paths interact, this becomes evermore difficult as the complexity increases. As a result, McCabe (1976, 314) recommends restructuring any code with a cyclomatic complexity greater than ten (an idea I explore further in chapter 6). This limit has been reaffirmed by many studies including the United State's National Institute of Standards and Technology who write, "the original limit of 10 as proposed by McCabe has significant supporting evidence" (Watson and McCabe 1996, sec. 2.5).

Applying Quantitative Metrics

Quantitative metrics lend themselves to statistical analysis. If a collection of code samples are each quantitatively measured, the measurements can be aggregated together and analysed to help identify general trends in the sampled population. This type of analysis does not appear to have been performed in any previous study on parametric models. As a result, the current understanding of parametric modelling is largely confined to firsthand accounts of working with specific parametric models (referred to in chapter 2.3). This leaves significant gaps in the understanding of parametric modelling and many basic questions – such as what is the average size of a parametric model or how complicated is the typical parametric model – remain unanswered. In the following pages I attempt to answer some of these basic questions and establish baselines for the key quantitative metrics I previously discussed (parts of this study were first published in Davis 2011b and then subsequently in Davis et al. 2011b).

Assembling a representative collection of parametric models is difficult since researchers generally only have access to parametric models created by themselves or their colleagues – a likely reason no previous study has quantitatively analysed a group of parametric models. But recently, with the advent of websites enabling communities of designers to share parametric models publicly, large collections of parametric models have been made available. One such website is McNeel's Grasshopper online forum (grasshopper3d.com) where, between 8 May 2009 and 22 August 2011, 575 designers shared 2041 parametric models. The models are all created in the Grasshopper modelling environment and tend be either a model a designer is having problems with or a model a designer thinks will solve another's problem. While this collection is not strictly representative of parametric modelling generally, it is a significant advancement over any previous study to be able to analyse, for the first time, how a large number of designers organise models created in a popular parametric modelling environment.

Method

To analyse the models publicly shared on the Grasshopper forum, I first download the 2041 parametric models. The oldest model was from 8 May 2009 and created with Grasshopper 0.6.12, and the most recent model was from 22 August 2011 and created with Grasshopper 0.8.0050. All the

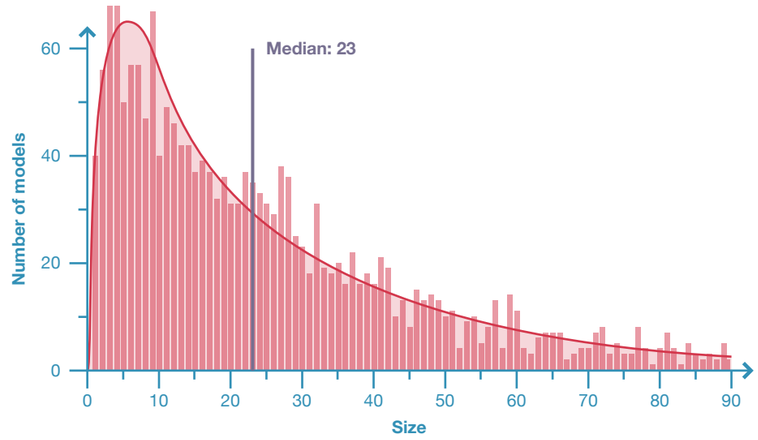


Figure 19: Distribution of model size in population of 2002 parametric models.

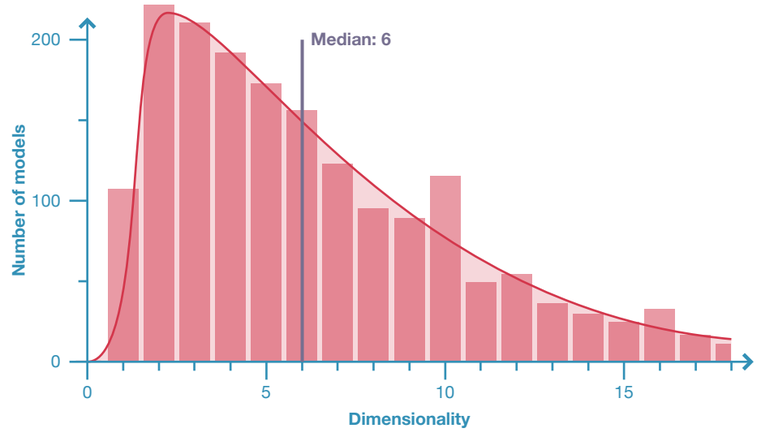


Figure 20: Distribution of model dimensionality in population of 2002 parametric models.

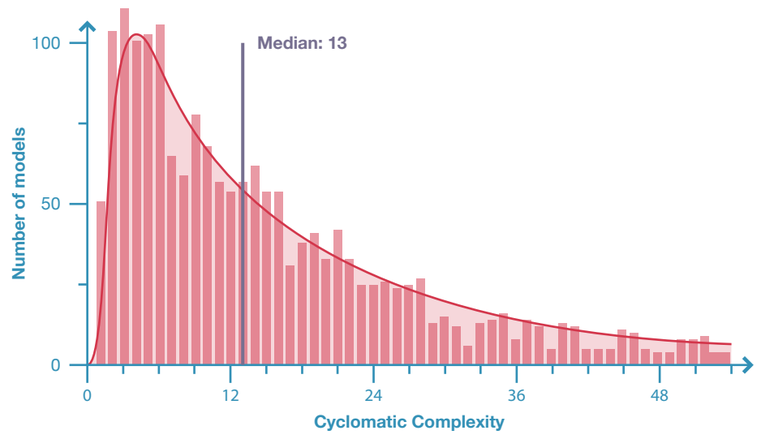


Figure 21: Distribution of model cyclomatic complexity in population of 2002 parametric models.

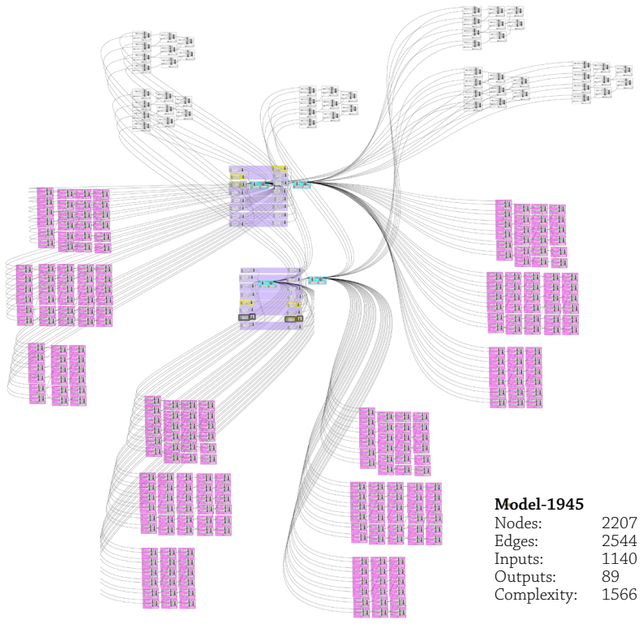


Figure 22: Model-1945, the largest and most complicated model in the sample. With over one thousand inputs, changing any part of the model is a guessing game. I have written previously (Davis 2011b) about how complexity can be reduced in this particular model by refactoring the duplicated elements and condensing the inputs into just twenty critical factors.

models were uploaded to the forum in the proprietary .ghx file format. I reverse engineered this format and wrote a script that extracted the parametric relationships from each file and parsed them into a directed acyclic graph. Thirty-nine models were excluded in this process, either because the file was corrupted or because the model only contained one node (which distorted measurements like cyclomatic complexity). The graphs of the remaining 2002 models were then each evaluated with the previously discussed quantitative metrics. The measurements were then exported to an Excel spreadsheet ready for the statistical analysis. In the analysis I have favoured using the median since the mean is often distorted by a few large outliers. Each of the key quantitative metrics is discussed below.

Size

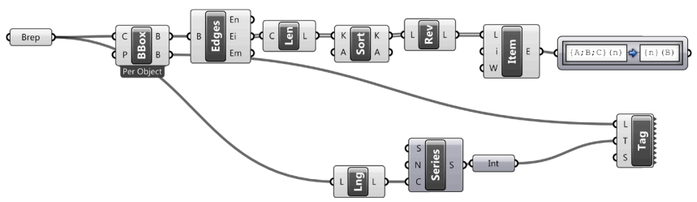
The sizes of the 2002 sampled Grasshopper models vary by a number of orders of magnitude; the smallest model contains just two nodes while the largest model contains 2207 nodes (fig. 22). The distribution of sizes has a positive skew (fig. 19) with the median model size being twenty-three nodes. I suspect the skew is partly because many of the models uploaded to the Grasshopper forum are snippets of larger models. The median may therefore be slightly higher in practice. Even with a slightly higher median,

Figure 23: Table of the most commonly used node types. These ten node types account for 40% of the 93,530 nodes contained within the 2002 sampled models.

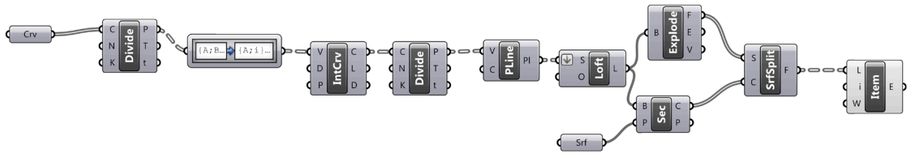
Rank	% nodes	Name	Function
1	12.6	Number Slider	Select numeric value
2	7.4	Panel	Read/write text
3	4.8	List Item	Select item in list
4	2.5	Point	Import point
5	2.4	Curve	Import curve
6	2.3	Line	Import line
7	2.3	Move	Move geometry
8	1.8	Scribble	Draw on graph
9	1.7	Series	Create series of numbers
10	1.6	Point XYZ	Create a point

the Grasshopper models (including the three models that contain more than one thousand nodes) are very modest compared to those seen in the context of software engineering.

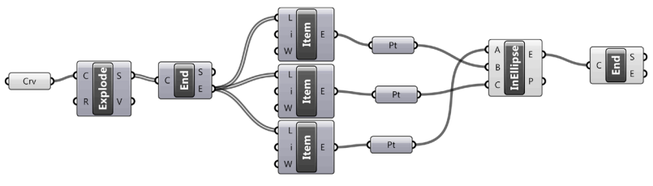
Given the numbers of nodes in a model, it is telling to see the typical function of these nodes. I took the 93,530 nodes contained within the 2002 Grasshopper models and ranked them based on function (the top ten are shown in figure 23). The most commonly used node was *Number Slider* [1], which is a user interface widget for inputting numeric values. Two more interface widgets are also feature highly on the list: *Panel* [2], which allows users to write and read textual data; and *Scribble* [8], which lets users explain a DAG by adding text. Also highly ranked were two nodes for managing data arrays: *List item* [3] and *Series* [9]. The fourth, fifth, and sixth most popular nodes are all ways of inputting geometry and managing the flow of data. The most popular node with a geometric function is *Move* [7], which is followed by *Point XYZ* [10]. In fact, only six of the twenty-five most popular nodes are geometric operations. This demonstrates that parametric modelling, at least within Grasshopper, is as much about inputting data, managing data, and organising the graph as it is about modelling geometry.



Model-660
Nodes: 12
Edges: 12
Inputs: 1
Outputs: 2
Complexity: 3

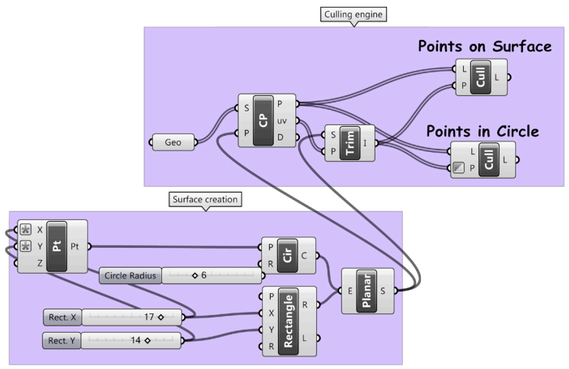


Model-1983
Nodes: 12
Edges: 12
Inputs: 2
Outputs: 1
Complexity: 3

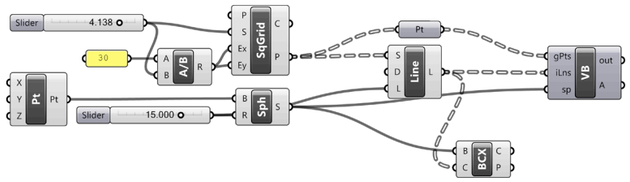


Model-2015
Nodes: 11
Edges: 12
Inputs: 1
Outputs: 1
Complexity: 3

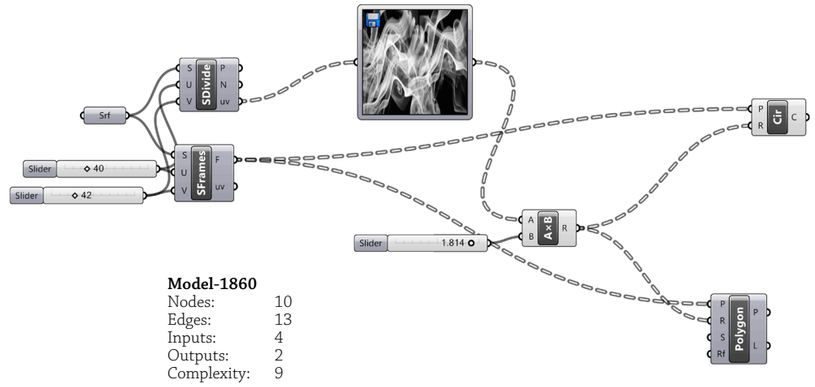
Figure 24: A comparison of models with different cyclomatic complexities. All six models are of a similar size and fairly representative of other models with equivalent complexities. This page: three simple models each with a cyclomatic complexity of three. Facing page: three slightly more complicated models with a cyclomatic complexity of either nine or ten.



Model-1913
Nodes: 12
Edges: 16
Inputs: 4
Outputs: 2
Complexity: 10



Model-313
Nodes: 11
Edges: 15
Inputs: 4
Outputs: 2
Complexity: 10



Model-1860
Nodes: 10
Edges: 13
Inputs: 4
Outputs: 2
Complexity: 9

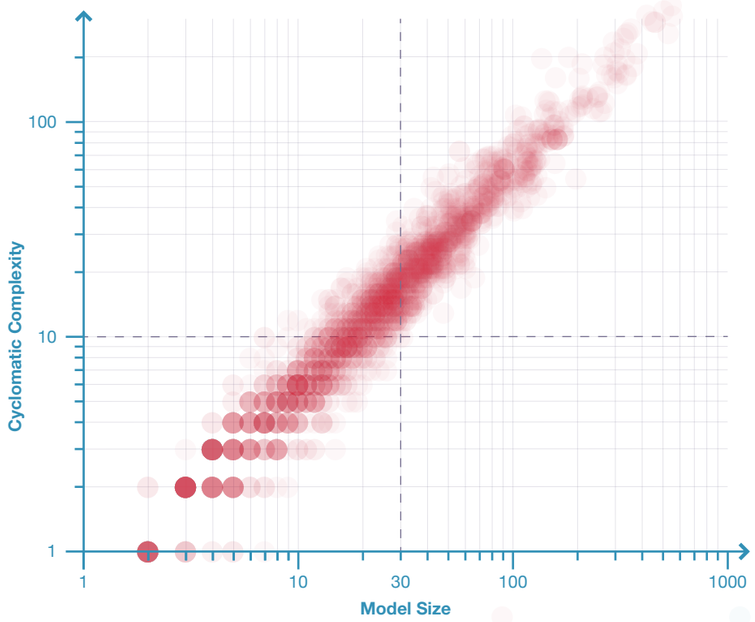


Figure 25: Model complexity plotted against size for 2002 parametric models. The distribution shows a strong correlation between size and complexity (r=0.98). The graph also shows that for models with more than thirty nodes it is inevitable they have a cyclomatic complexity greater than ten.

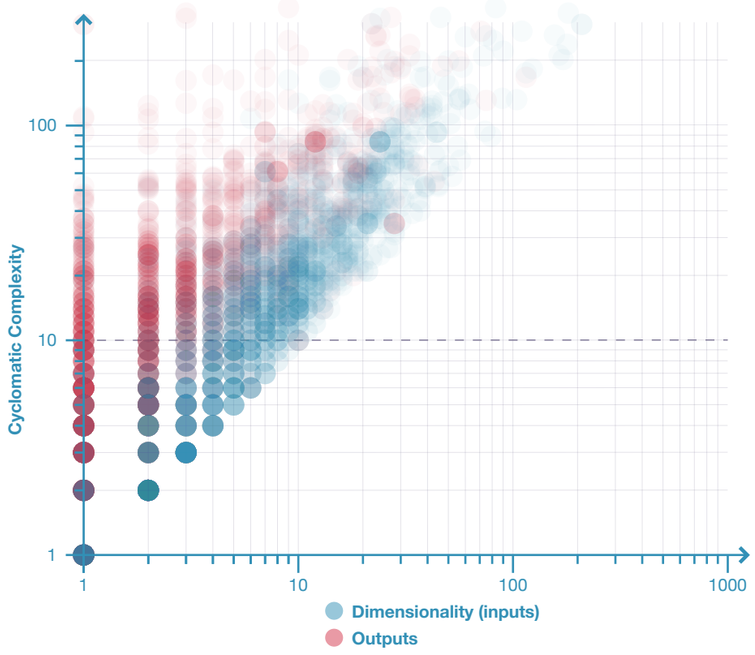


Figure 26:

Model complexity plotted against dimensionality for 2002 parametric models.

Model complexity plotted against the number of outputs for 2002 parametric models.

The distribution shows that the number of model outputs has little bearing on complexity (r=0.71) since for any given number of outputs there are a range of complexities associated (the vertical spread of red dots). In comparison, the number of inputs has a stronger (r=0.91) relationship to model complexity (the blue dots are more linear and less vertically spread).

Dimensionality

The vast majority of models have a similar dimensionally; 75% possess between one and eleven inputs with the median being six inputs (fig. 20). Seventeen outliers have more than one hundred inputs and the most extreme model contains 1140 inputs. When examining the models with a high dimensionality, it is strikingly difficult to understand what each input does and even more difficult to change the inputs meaningfully en masse (often the only way to guess and check). I suspect the comparatively low dimensionality shown in the vast majority of models may be because designers can only comfortably manipulate a few parameters at a time. Therefore, while parameters are a key component of parametric modelling (some would say the defining component: see chap. 2.1) the majority of designers use parameters sparingly in their models.

Cyclomatic Complexity

There is a high variance in the cyclomatic complexity of the sampled models. The median complexity is fourteen (fig. 21) but the range extends from simple models with a complexity of just one, to extremely complex models with a complexity of 1566 (fig. 22). Within this variance, 60% of models have a complexity greater than ten – the limit McCabe (1976, 314) suggested. The differences between complex and simple models are visually apparent in figure 24 where the two extremes are displayed side-by-side. In figure 24, the simple models have orderly chains of commands while the models with a higher cyclomatic complexity have interwoven lines of influence that obfuscate the relationships between nodes. This seems to indicate that cyclomatic complexity is effective in classifying the complexity of a parametric model.

A model's cyclomatic complexity and size are strongly correlated (r=0.98;[2] fig. 25). This correlation is significant because it indicates that while a parametric model can theoretically be both large and simple, in actuality, large models tend to be complex (all the models with more than thirty nodes had a cyclomatic complexity greater than McCabe's limit of ten). A similar correlation exists in software engineering. One such example is van der Meulen and Revilla's (2007, 206) survey of fifty-nine textual programs that found cyclomatic complexity and LOC to have a correlation of r=0.95.

2 r is Pearson's coefficient. A value of 1 indicates that two variables are perfectly correlated (all sampled points fall on a line), a value of 0 indicates that two variables are not not correlated in any way (one does not predict the other), and a negative value indicates an inverse correlation.

This correlation suggests that complexity is an inevitable by-product of size in both software engineering and parametric modelling. Similar relationships exist for a model's dimensionality (r=0.91) and outputs (r=0.71), although neither correlates with complexity to the same degree as a model's size (fig. 25 & 26).

Learning from Quantitative Data

This study is an important first step towards understanding the properties and variations of a typical parametric model. The 2002 Grasshopper models surveyed show that parametric models are generally small and complex. The average model contains twenty-three nodes and even the largest models, with just over one thousand nodes, are modest in the context of software engineering. The size of the model is highly correlated with the model's complexity, which tends to be very high overall. While one may intuitively expect that the majority of a parametric model consists of parameters and geometry, this study shows that organising the graph and managing data are often the most common components of parametric models created in Grasshopper. Parameters tend to be used surprisingly sparingly, with the vast majority of models only containing between one and eleven parameters.

Another important outcome from this survey is the validation of the quantitative metrics. The study demonstrates that nodes are a good proxy for a model's size and that the dimensionality can reveal unintuitive insights regarding the use of parameters in parametric models. Furthermore, the cyclomatic complexity seems to fairly accurately differentiate between simple and complex models. However, despite the validation of these quantitative metrics, they still only tell a narrow part of a model's story; a story that can be further triangulated with qualitative measures.

4.4 Qualitative Flexibility

Many aspects of parametric flexibility elude quantitative measurement. While it is useful to know the size of a model or the complexity of a model, by themselves, these measurements give an incomplete picture. Bertrand Meyer (1997, 3) argues, in his seminal book *Object-Oriented Software Construction*, "software quality is best described as a combination of several factors." Meyer (1997, chap. 1) spends the first chapter of his book expounding the following ten factors of "software quality":

1. **Correctness:** ability of software products to perform their exact tasks, as defined by their specification.
2. **Robustness:** the ability of software systems to react appropriately to abnormal conditions.
3. **Extendability:** the ease of adapting software products to changes of specification.
4. **Reusability:** the ability of software elements to serve for the construction of many different applications.
5. **Compatibility:** the ease of combining software elements with others.
6. **Efficiency:** the ability of a software system to place as few demands as possible on hardware resources.
7. **Portability:** the ease of transferring software products to various hardware and software environments.
8. **Ease of use:** the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems.
9. **Functionality:** the extent of possibilities provided by a system.
10. **Timelessness:** the ability of a software system to be released when or before its users want it.

While other authors have constructed similar lists of software quality (Meyer 1997, 19-20), Meyer's list holds significant cachet in software engineering because it belongs to one of the most cited books in computer science.[3] There are notable correlations between Meyer's list and the ISO/IEC standard for *Software Product Quality* (ISO 2000), with Meyer's efficiency, portability, ease of use, and functionality being word-for-word identical to the ISO categories. In this thesis I take the factors

3 CiteSeer (2012) say *Object-Oriented Software Construction* is the sixty-second most cited work in their database of over two million computer science papers and books.

Meyer identifies as being crucial to software quality and I use them as a structure for qualitative evaluations of parametric models. In particular, I make reference to Meyer's concepts of correctness [1], extendability [3], reusability [4], efficiency [6], ease of use [8], and functionality [9], which I will now briefly explain in more detail.

Correctness

Correctness concerns whether software does what is expected. In some circumstances correctness is obvious; if you create a parametric model to draw a cube, the model is correct if it draws one. But in most circumstances correctness is non-trivial since it can be difficult to determine what is expected and to ensure this happens through a range of input parameter values. Software engineers have developed a range of methods for ascertaining whether software is correct – unit testing being one notable example. While I suspect architects would benefit from adopting these practices, this is a large area of research outside the scope of my thesis (as discussed in chapter 3). For the remainder of this thesis I have used correctness to denote that a parametric model is free from any major defects; it is not creating spheres when it should be creating cubes.

Extendability

Extendability is essentially a synonym for flexibility; the ease with which software adapts to changes. Meyer (1997, 7) says extendability correlates with size, since "for small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt." This notion corresponds to what other authors have written about the software crisis (see chap. 3.1) and it corresponds to the relationships between software size and cyclomatic complexity that I have empirically shown. Meyer (1997, 7) goes on to argue that extendability can be improved by ensuring the code has a "simple architecture," which can be achieved by structuring the code with "autonomous modules." While I explore extendability throughout this thesis, I pay particular attention to the structure of parametric models in chapter 6.

Reusability

Reusability pertains to how easily code can be shared, either in part or in whole. Meyer (1997, 7) notes that "reusability has become a pressing concern" of software engineers. As I have shown in chapter 2.3, the reusability of parametric models is also a concern of many architects.

Efficiency

Efficiency describes how much load a program places on hardware. This is particularly pertinent to architects because a program's efficiency helps determine its latency, which, in turn, affects change blindness (see chap. 2.3). In extreme cases the model's efficiency may even determine its viability, since certain geometric calculations are so computationally demanding that inefficient models can slow them to the point of impracticality. However, Meyer (1997, 9) tells software engineers "do not worry how fast it is unless it is also right" and warns, "extreme optimizations may make the software so specialized as to be unfit for change and reuse." Thus, efficiency can be important but it needs to be balanced against other attributes like correctness and reusability.

Ease of Use

Ease of use is fairly self-explanatory. For architects, ease of use applies to both the modelling environment and the model. A modelling environment's ease of use concerns things like user interface and modelling workflow. A designer familiar with a modelling environment will tend to find it easier to use, which impacts how fast they can construct models (construction time) and how competently they can make changes (modification time). In addition to the modelling environment being easy to use, the model itself needs to be easy to use. I have spoken previously about the importance of dimensionality and complexity when it comes to understanding and changing a model. Meyer (1997, 11) echoes this point, saying a "well thought-out structure, will tend to be easier to learn and use than a messy one."

Functionality

Functionality to Meyer (1997, 12) denotes "the extent of possibilities provided by a system." Like ease of use, functionality is applicable to both the modelling environment and the model. The key areas of functionality in a modelling environment include the types of geometry permissible, the types of relationships permissible, and the method of expressing relationships. Modelling operations that are easy to implement in one environment may be very difficult (or impossible) in another due to variations in functionality. Likewise, changes easily permissible in one environment may be challenging in another. Therefore, the functionality of a modelling environment helps determine the functionality of the parametric model. This is a determination that often comes early in the

project since changing the modelling environment mid-project normally means starting again.

Using Qualitative Metrics

All of the qualitative metrics require some form of consideration and judgment in their application. Some have established protocols of observation – for example, there are well-researched ways to conduct usability studies in order to analyse ease of use. Other qualitative assessments can be logically deduced through comparisons – for example, the functionality of a modelling environment can be evaluated by comparing its features to those of other environments. But other attributes, like extendability, fall upon expert judgment to analyse. None of these are definitive measurements, for even the quantitative measurements are distorted by what they cannot measure. However, the qualitative measurement do provide a vocabulary of attributes to begin capturing the qualities of a parametric model.

4.5 Conclusion

Meyer (1997, 15) stresses that software metrics often conflict. Each metric offers one perspective, and improvements in one perspective may have negative consequences in another. For example, making a model more efficient may make it less extendible, and making a model more reusable may harm the latency. Furthermore, measured improvements may not necessarily manifest in improved flexibility since flexibility is partly a product of chance and circumstance; an apparently flexible model (one that is correct, easy to use, and with a low cyclomatic complexity) can stiffen and break whilst a seemingly inflexible model may make the same change effortlessly. This uncertainty makes any single measure of flexibility – at best – an estimation of future performance.

To help mitigate the biases of any single metric, I plan to aggregate a triangulated perspective of the case studies using a variety of metrics. In this chapter I have discussed a range of metrics applicable to parametric modelling: from quantitative metrics to measure time, size, and complexity; to qualitative metrics to begin discussing qualities like correctness, functionality, and reusability. By gathering these measurements together in this chapter I have begun to articulate a vocabulary for discussing parametric models; a vocabulary that goes beyond the current binaries of failure and success. Using parts of this vocabulary I have been able to analyse, for the first time, a large collection of parametric models in order to get a sense of the complexity, composition, and size of a typical parametric model. This demonstrates the viability of quantitatively measuring qualities like cyclomatic complexity but also demonstrates why quantitative metrics alone are not enough to observe the case studies.

In addition to the suite of metrics, this chapter has also identified three case studies to test various aspects of the software engineering body of knowledge. The case studies have been selected not because the cases are necessarily representative of challenges architects typically encounter, but because cases provide the best opportunity to learn about these challenges. Each of the following three chapters contains one of these case studies and makes use of a variety of the metrics discussed in this chapter.

Case studies

The following three chapters document a case study each. The case studies take the software engineering concepts discussed in chapter 3.2 and apply them to architecture projects where their performance is measured using the research instruments discussed in chapter 4.

	Case A
Chapter:	5
Main Subject:	Logic Programming
Main Project:	Realignment of the Sagrada Família frontons

	Case B
Chapter:	6
Main Subject:	Structured Programming
Main Project:	Designing Dermoid

	Case C
Chapter:	7
Main Subject:	Interactive Programming
Main Project:	Responsive Acoustic Surfaces & The FabPod

5 Case A: Logic Programming

Project: Realignment of the Sagrada Família frontons.

Location: Barcelona, Spain.

Project participants: Daniel Davis, Mark Burry, and the Basílica i Temple Expiatori de la Sagrada Família design office.

Related publication:

Davis, Daniel, Jane Burry, and Mark Burry. 2011. "The Flexibility of Logic Programming." In *Circuit Bending, Breaking and Mending: Proceedings of the 16th International Conference on Computer Aided Architectural Design Research in Asia*, edited by Christiane Herr, Ning Gu, Stanislav Roudavski, and Marc Schnabel, 29–38. Newcastle, Australia: The University of Newcastle.

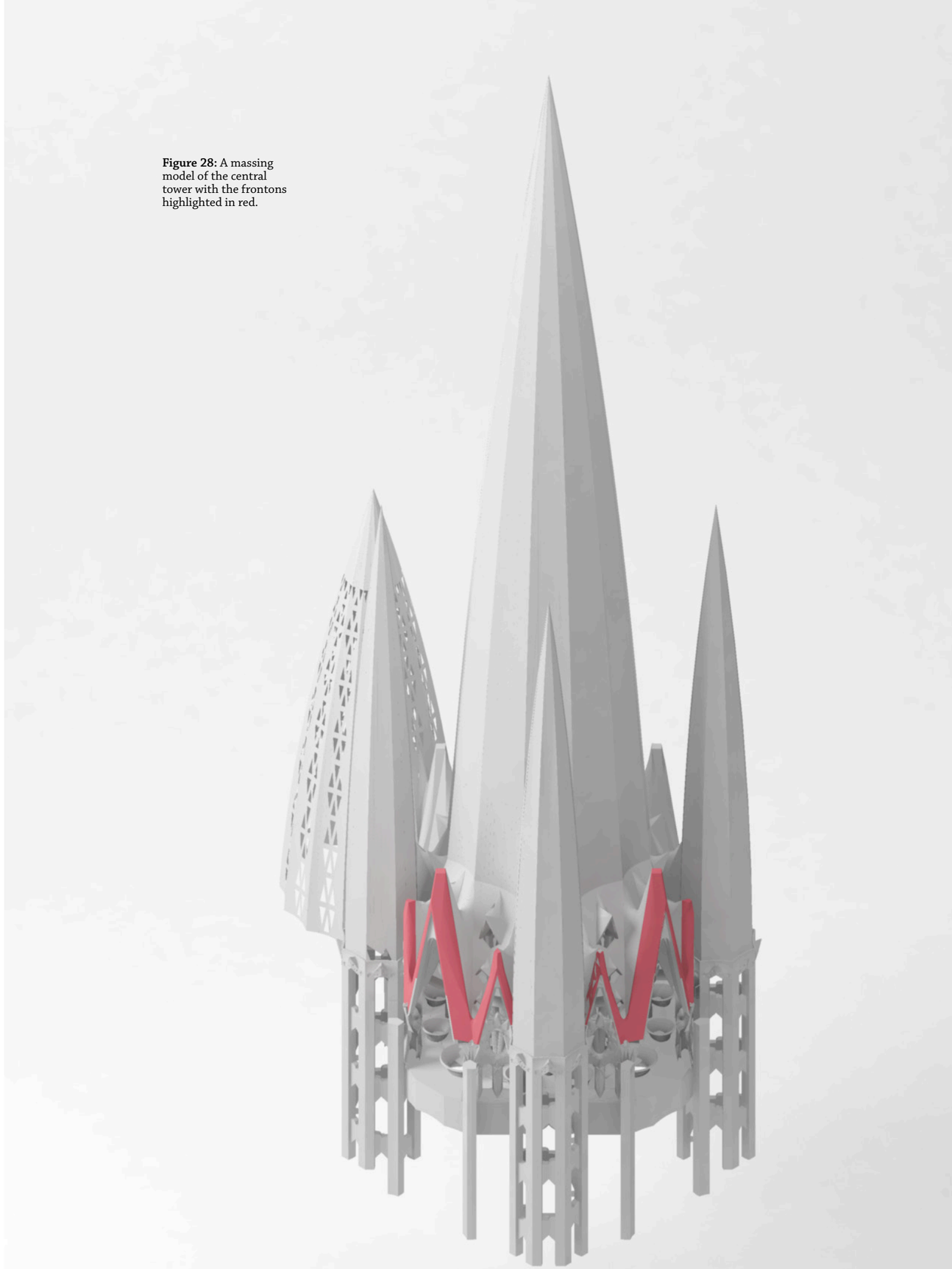
Figure 27: Lluís Bonet i Garí's interpretation (circa 1945) of what the Sagrada Família would look like when completed. The tall central tower capped by a cross is dedicated to Jesus Christ.

5.1 Introduction

High above the main crossing of Basílica de la Sagrada Família, Antoni Gaudí planned a tall central tower dedicated to Jesus Christ (fig. 27). The tower marks the basilica's apex of 170 metres and the culmination of a design that has been in development for over one hundred years. Today, almost eighty-five years after Gaudí's death in 1926, the team of architects continuing his work is preparing to construct the central tower.

At the base of the tower sit three matching gabled windows, each seventeen metres high (fig. 28). The stone head to these windows is called a *fronton* by the project team (*fronton* being Catalan for *gable*). The fronton design,

Figure 28: A massing model of the central tower with the frontons highlighted in red.



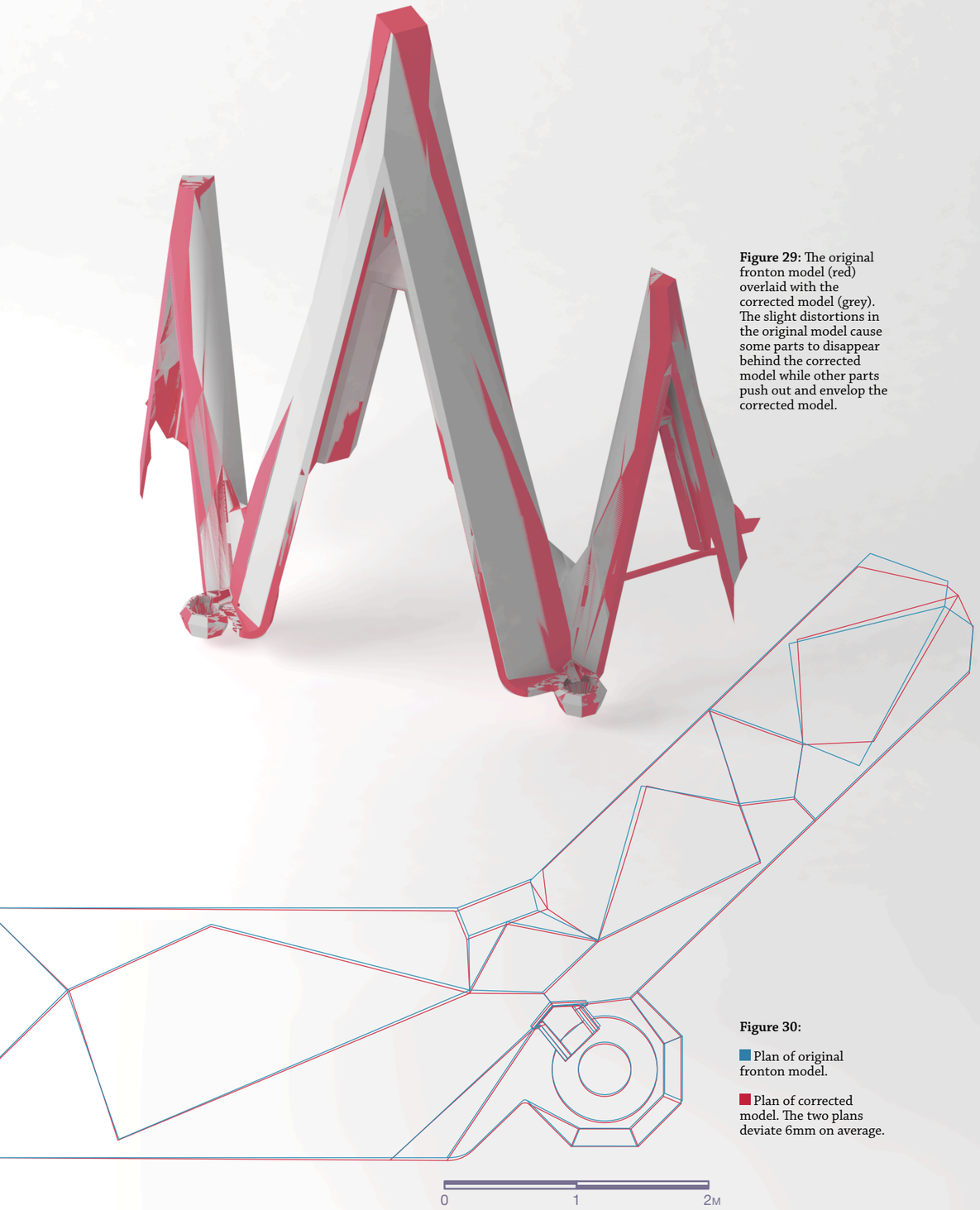


Figure 29: The original fronton model (red) overlaid with the corrected model (grey). The slight distortions in the original model cause some parts to disappear behind the corrected model while other parts push out and envelop the corrected model.

Figure 30:

Plan of original fronton model.

Plan of corrected model. The two plans deviate 6mm on average.

like everything else on the church, has stretched over a period of years. The progression of software in this time has seen the digital fronton model pass from one CAD version to the next. At some stage during this process, the parametric relationships in the fronton model were removed and the model became explicit geometry. In 2010, as the project team were preparing the fronton's construction documentation, they came across a curious problem with the model: passing the model between software had caused slight distortions. With the parametric relationships removed, fronton faces that should have been planar contained faint curves, lines that should have been orthogonal were a couple of degrees off, and geometry that should have been proportional was just a touch inconsistent (fig. 29 & 30). None of these distortions were apparent at first glance and on a more routine project they probably would not be of concern. However, on a project as meticulous as the Sagrada Família – where the design has germinated for decades – it was vitally important to remove any imperfections, or at least get them within the tight tolerances of the seven-axis robot scheduled to mill the stone. My task was to straighten the explicit geometry in the fronton model by converting it back into a parametric model and reasserting the original parametric relationships.

The realignment of the frontons presents a unique parametric modelling case study. Besides contributing to one of the earliest examples of parametric architecture, the project demands an extraordinary level of precision, while the ambiguity of what constitutes *straightened* requires collaboration with team members in Melbourne and Barcelona. This case study is not necessarily representative of typical parametric architecture projects but, as I discussed in chapter 4.1, the project's unique circumstances offer what Robert Stake (2005, 451) calls "the opportunity to learn." In this chapter I observe how the demands of the fronton realignment manifest within two parametric models. In particular, I examine how the language paradigm of the parametric model affects its behaviour (following on from chapter 3.2, where I noted most parametric models are based on a narrow range of language paradigms). I begin this chapter by discussing the taxonomy of possible language paradigms and identifying how the rarely used logic programming paradigm may be applicable to parametric modelling. I then twice realign the frontons, once with a conventional dataflow parametric model, and once with a parametric model based on logic programming. The differences between these two language paradigms form the discussion of this chapter.

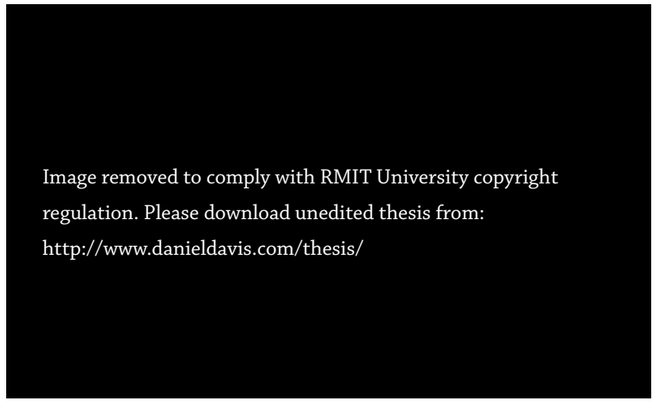


Figure 31: Van Roy's photograph of the Sagrada Família on the cover of *Concepts, Techniques, and Models of Computer Programming* (Van Roy and Haridi's 2004). The Japanese edition uses a slightly different photograph of the Sagrada Família.

5.2 Programming Paradigms

The Sagrada Família graces the cover (fig. 31) of Peter Van Roy and Seif Haridi's (2004) seminal book on programming languages entitled *Concepts, Techniques, and Models of Computer Programming*. Van Roy took the photograph and chose it for the cover because "the still unfinished Expiatory Temple of the Sagrada Família in Barcelona is a metaphor for programming" (Van Roy, n.d.). Van Roy and Haridi never foresaw, however, that the opposite maybe true, that the programming paradigms they discuss in their book could help designers of the church on its cover. In the preface Van Roy and Haridi (2004, xx) allude to another architect, Mies van der Rohe, in a section about programming paradigms titled "more is not better (or worse), just different." A programming paradigm in this context is the set of underlying principles that shape the style of a programming language. For Van Roy and Haridi, these styles are not better nor worse, just different to one another. It is this difference that I consider in relation to the parametric models of the Sagrada Família.

Programming paradigms are roughly divided by Van Roy and Haridi (2004) as well as others like Appleby and VandeKopple (1997) into imperative paradigms or declarative paradigms (fig. 32). Imperative languages describe a sequence of actions for the computer to perform – much like imperative verbs in the English language. In contrast, declarative languages "define the what (the results we want to achieve) without explaining the how (the algorithms needed to achieve the results)" (Van Roy and Haridi

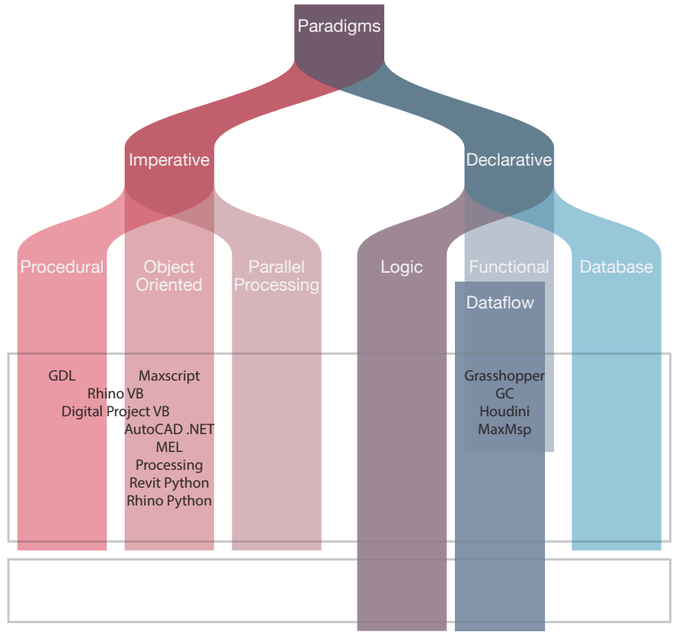


Figure 32: The programming languages architects use categorised by Appleby and VandeKopple's (1997, xiv) taxonomy of programming paradigms.

The two paradigms explored in this case study.

2004, 114). Imperative and declarative languages can be further classified into more specific paradigmatic subcategories, as shown in figure 32. Most programming languages are based on at least one of these subcategories, and many spread out to embody multiple paradigms within the one language – more is not better (or worse).

As discussed in chapter 3.2, the languages favoured by designers tend to occupy a narrow range of possible paradigms (fig. 32). The major textual CAD programming languages are all predominantly imperative[1] with a bias towards procedural imperativeness. This is not surprising considering that the world's five most popular programming languages on the TIOBE (2012) index are also predominantly imperative[2] (although perhaps more spread out on the imperative spectrum). In contrast, visual programming languages tend towards declarativeness. The major visual CAD programming languages all reside in a very narrow subsection of

1 These include: 3dsMax: Maxscript; Archicad: GDL; Digital Project: Visual Basic; Maya: Maya Embedded Language; Processing: Java; Revit: Visual Basic & Python; Rhino: Visual Basic & Python; Sketchup: Ruby.
2 As of May 2012 the worlds five most popular programming languages, as measured by TIOBE (2012), are: C, Java, C++, Objective-C, and C#.

declarative programming known as dataflow programming.[3] In this chapter I compare two declarative paradigms – dataflow programming and logic programming – as a means to construct the parametric models of the Sagrada Família frontons.

5.3 Challenges of Dataflow

In the introduction to *Data Flow Computing*, John Sharp (1992, 3) defines a dataflow program as "one in which the ordering of operations is not specified by the programmer, but that is implied by the data interdependencies." In other words, a dataflow language describes the connections between computational operations, which is different to the imperative approach of listing operations in the order they should occur. When a dataflow program is run, the computer infers the precise order of operations from the stated connections between operations.

The quintessential example of a dataflow program is a spreadsheet. The user of a spreadsheet specifies how cells connect and how cells should process data but leaves the computer to decide the precise order in which cells get updated. The same principle applies to certain parametric modelling software, like Digital Project. Users of this software specify a network of connections between geometric operations while the computer manages the exact sequencing and execution of these operations. Many visual programming languages operate on a similar principle since the connections between operations can be represented using a type of flow-chart know as a Directed Acyclic Graph (DAG). The two components of a DAG are nodes and directed edges (fig. 33). In a visual program a node represents an operation and a directed edge represents a connection (a flow of data between two operations), which is how the visual programming environments used by architects (Grasshopper, GenerativeComponents, and Houdini) represent dataflow programs.[4]

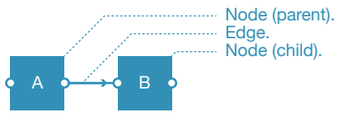


Figure 33: The parts of a DAG.

3 These include: Grasshopper; GenerativeComponents; Houdini; and MaxMsp.
4 I discuss, in chapter 7, an alternative way to represent a dataflow program textually rather than visually.

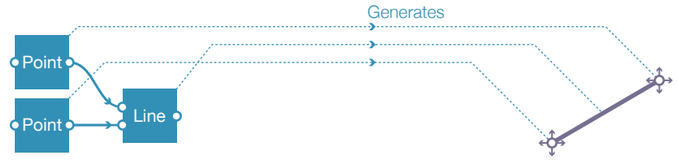


Figure 34: The relationship between a DAG and the geometry it generates. In the DAG the line is a child of the two points, accordingly, the line's geometry depends entirely on the location of the points. Moving the geometry of either point would also move the line.

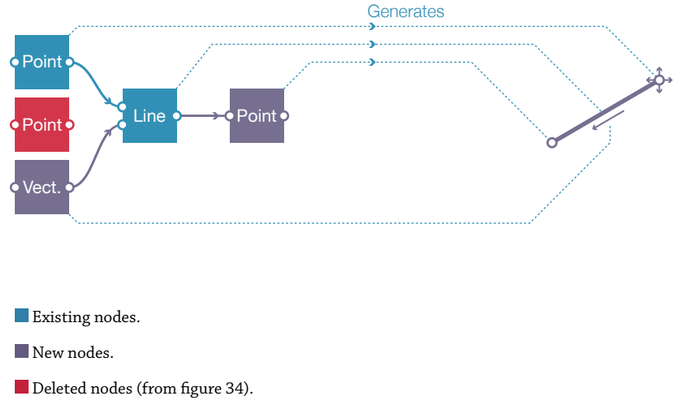


Figure 35: Modifications to the DAG from figure 34. The geometry is the same but the connections have been changed: one of the points is now a child of the line. In the geometric model, the child point can no longer be moved directly since its location now depends on the line's position (the parent of the point). While the geometry is the same as figure 34, the change in hierarchy requires adding and removing a number of nodes.

A critical component of dataflow programming is that the flow of data has a direction specified by the programmer. In defining the direction of data, the source of data is termed the parent and the receiver of data is termed the child (fig. 33). The direction has important implications since the programmer is not just specifying that operations are connected but also necessarily giving a hierarchy to these connections. As a consequence, if the programmer reverses the flow of data – turning children into parents and parents into children – they risk disrupting the program's structure. For example, the dataflow program in figure 34 generates a line between two parent points. If the data in this model is reversed (a point becomes a child of the line) the change requires both deleting nodes and adding new nodes (fig. 35). Even though the changes introduce no new geometry, they agitate the hierarchy so much that starting over would be as easy as changing the model in figure 34 to match the model in figure 35. These disruptions to the hierarchy would be avoidable if the designer did not need to specify the direction of connections and instead only needed to specify that two things are connected. In this case study I consider whether logic programming can help remove the need to specify the direction between operations in much the same way dataflow programming removes the need to specify an order of operations.

5.4 Logic Programming

Logic programming, like dataflow programming, fits into the declarative branch of programming paradigms (fig. 32). Defined by Sterling and Shapiro (1994, 9), the first part of "a logic program is a set of axioms, or rules, defining relations between objects." These relations do not specify flows of data but rather express, in first order logic, statements of fact. For example, an axiom might be: `A cat is an animal`. The second part of a logic program is an interpreter that reads the axioms and logically deduces the consequences (Sterling and Shapiro 1994, 9).[5] Using the above axiom, the interpreter might be asked `what is an animal?` to which it would deduce: `cat`.

At the genesis of logic programming, in the late 1960s and early 1970s, many expected logic programming's formalisation of human reasoning to push humanity beyond its cognitive limitations (limitations that had ostensibly brought about events like software crisis). Robert Kowalski (1998) recalls his contemporaries during this period employing logic programming for ambitiously titled projects involving "natural language question-answering" (38) and "automated theorem proving" (39). The initial developments were promising with researchers discovering various ways to get computers to seemingly reason and respond to a series of questions, even if the questions were always confined to small problem domains (Hewitt 2009). The hope was that if a computer could answer a simple question like *how many pyramids are in the blue box?* then it would be possible to get a computer to answer a more difficult question like *What does "cuantas pyramides se encuentran dentro de la caja azul" mean?* But, as I have discussed in chapter 3, systems addressing small problems – be they computer programs or parametric models – do not always scale to address larger problems. The initial interest in logic programming waned as success with small, well-defined problems failed to bring about widespread success with larger problems. Today the most common logic programming

5 On the surface, there may appear to be many examples of logic programming used in CAD. For example, Sketchpad (Sutherland 1963), has a geometric constraint solver that allows users to define axioms between geometry. However, Sketchpad does not satisfy the definition of logic programming since the axioms are interpreted using numeric algorithms and an early form of hill climbing rather than through logical deduction (Sutherland 1963, 115-19). The logical deduction that forms the basis for logic programming was not invented until six years after Sketchpad.

language – Prolog – only ranks thirty-sixth on the TIOBE (2012) index of popular programming languages. Nevertheless, logic programming still finds niche applications in expert reasoning – particularly expert reasoning about relationships (for instance, IBM's Jeopardy winning computer, Watson, was partly based on Prolog [Lally and Fodor 2011]).

In the architecture industry, logic programming followed a similar arc, albeit a few years behind what was happening in software engineering. Architectural researchers took the work done on spatial logic programming and enthusiastically applied it to a favourite problem of the time: room layout (Keller 2006). A paper typical of the period is Peter Swinsons' (1982) optimistically named *Logic Programming: A Computing Tool for the Architect of the Future*. In this paper Swinson (1982, 104) demonstrates how Prolog can solve four different layout problems before concluding "this new way of programming does indeed hold much promise for the future" (a similar method is used by Márkusz [1982]). A flurry of interest followed in the 1980s.[6] In 1984, Gonzalez et al. demonstrated two-dimensional drafting using logic programming, which they found "more concise, more reliable, and clearer" as well as "more efficient" than imperative programming (Gonzalez et al. 1984, 74). Further examples of two-dimensional shape drawing were given by Brüderlin (1985) and subsequently Helm and Marriott (1986). A more comprehensive attempt to generate three-dimensional models was made by Robert Woodbury (1990) using the ASCEND language, which is not strictly a logic programming language although it shares many similarities. The zenith for logic programming in architecture was arguably reached the same year with the publication of William Mitchell's (1990) *The Logic of Architecture*.

Despite *The Logic of Architecture*'s success, it marks the beginning of the end. While Mitchell was able to apply his ideas retrospectively to Palladian villas, he never tested them on real architecture projects. Indeed, none of the papers I have referenced (or that I can find) discuss using logic programming on real architecture projects – even though many of them, like Swinson, were making confident proclamations that logic programming would become the "computing tool for the architect of the future" (Swinson 1982). Much like the software engineers that came before them, these researchers all tested logic programming with small, idealised problems

6 For a complete history see: Fudos 1995.

assuming the success of these small systems were representative of future successes at larger scales. The initial interest in logic programming waned (like it had done in software engineering a few years earlier) as success with small, well-defined problems failed to bring about widespread success with larger problems. Nevertheless, there are still a couple of recent examples of logic programming being used by researchers (still on small, idealised problems) including Martin and Martin's PolyFormes tool (1999) and Makris et al. MultiCAD (2006). However, in contrast to the widespread imperative programming paradigm used by architects, logic programming never quite became the computing tool for the architect of the future. In fact, I can find no example of logic programming ever being used on a real architecture project.

The following case study differs from prior logic programming research in three important ways. Firstly, it investigates logic programming in the context of a real design problem from the Sagrada Família rather than discussing logic programming theoretically applied to an idealised problem. Secondly, it focuses on deducing the hierarchy of relationships for a parametric model instead of directly generating geometry, or laying out rooms, or drawing Palladian villas. Finally, this research does not aspire to develop the computing tool for the architect of the future; instead it aims to observe how programming paradigms influence the flexibility of parametric models.

5.5 Logic Programming Parametric Relations

As part of my research I developed a logic programming language for generating parametric models. A designer using the language can specify connections between operations, which are then used by an interpreter to derive the flow of data without the designer needing to specify a hierarchy (like they would in a dataflow language). I elected to develop my own logic programming interpreter in C# rather than using an existing logic programming implementation. This allowed me to link the interpreter directly to a parametric engine I had already built on-top of Rhino 4's geometric API (an engine reused for the case study in chapter 7).

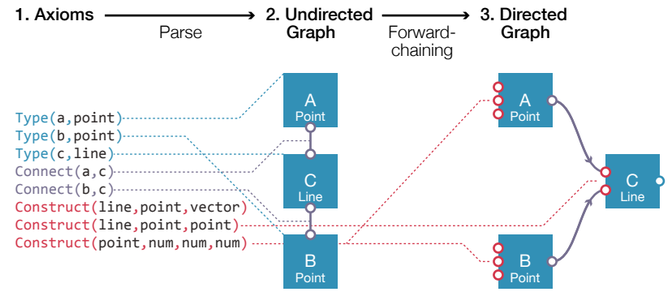


Figure 36: The major stages involving in deducing a parametric model to satisfy a set of textual axioms. The dashed lines show how the axioms generate particular parts of the graphs in the various stages.

My logic programming interpreter uses a three-stage process to derive a parametric model automatically from a set of axioms. These stages are illustrated in figure 36. The first stage is to read the text file containing the axioms. In the second stage, the axioms are parsed into an undirected graph (using rules I will explain shortly). At the final stage, the interpreter uses forward-chaining to deduce the directness of the graph, which produces the DAG of a parametric model satisfying the initial axioms.

There are three types of axioms permitted in the language:

Axiom Type 1: Geometric

A geometric axiom defines a geometric entity's unique name and geometry type. For example, a line with the name of *C*, is defined by the axiom: `type(C,line)`.

Axiom Type 2: Connection

A connection axiom establishes a connection between two geometric entities. For example, to connect line *C* to point *B*, the axiom would be: `connect(B,C)`. Connection axioms do not define the direction of the connection, they only state that two geometric entities are related.

Axiom Type 3: Construction

A construction axiom describes a combination of parents that define a particular geometry type. For example, a line can be defined by two parent points, which is expressed with the axiom: `construct(line,point,point)`. Any particular geometry type can have multiple construction axioms. For example, a line may also be defined by a point and a vector: `construct(line,point,vector)`. Using forward-chaining, the interpreter selects the most appropriate construction axiom for a given situation. In figure 36, the forward-chaining algorithm has inferred that node *C*, a type of line, must be defined by two parent points. To satisfy this relationship, the interpreter organises the undirected graph into a hierarchy so that line *C* becomes the child of the two points (*A* & *B*). The following page explains the three rules used to infer the most appropriate construction axioms.

Figure 37: *A* must be the parent of *C*, since none of the construction axioms for *A* require a line as a parent, whereas both the construction axioms of *C* require a point as a parent.

Inference 1: Asymmetric constructors

The construction axioms list the type of geometry that can act as the parent for any given type of node. If two nodes are connected, and the first node has a construction axiom allowing the second node to be its parent, and the second node does not have any construction axioms that allow the first node to be its parent, then the first node must necessarily be the parent of the second node.

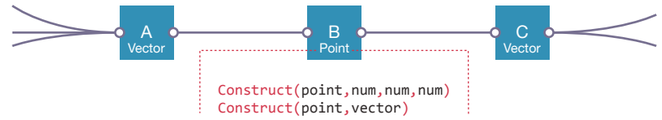


Figure 38: The first construction axiom requires *B* to have three parents that are numbers. Since *B* is not connected to any numbers, it will never be able to fulfil this axiom and therefore the axiom can be eliminated. The second axiom is still possible, which means one of the vectors much be a parent to *B*.

Inference 2: Constructor elimination

If a node has multiple construction axioms, some axioms can be eliminated if the node is not connected to the right combination of node types to fulfil a particular construction axiom. Eliminating constructor axioms makes it more likely to find asymmetric constructors.

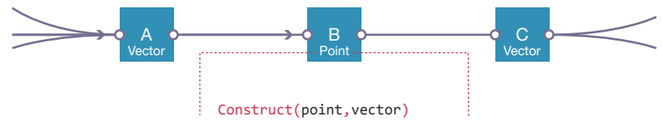


Figure 39: If *A* is *B*'s parent, then *B*'s construction axiom is fulfilled and it cannot have anymore parents. Accordingly, all the remaining undirected connections must flow away from *B*, and thus *C* is a child of *B*.

Inference 3: Constructor definition

If a node has all the required parents to fulfil a construction axiom, then it does not need any more parents and all the remaining nodes connected through undirected connections must be children. Once this rule becomes relevant the directedness normally cascades through the graph as large numbers of connections become directed.

Ideally these three rules are enough to deduce the entire flow of the graph. On occasion the axioms will define an over-constrained graph, producing a situation where two nodes are connected but neither is a permissible parent of the other. In this case the interpreter first looks to see if the addition of numerical nodes will allow progress past this impasse. If not, the user is asked to resolve the axiom conflicts.

A designer using this logic programming system only needs to define connections between geometric entities, which leaves the interpreter to infer the direction of the connections. Figure 40 shows how axioms transform into a parametric model and how changes to the axioms automatically result in new dataflows. The example in figure 40 is identical to the earlier dataflow example in figure 35 except, unlike the dataflow example, the changes to the parametric model's hierarchy are managed invisibly by the interpreter; the designer adds a few axioms and the new dataflow is automatically derived by the interpreter. The following section describes how these changes come to impact real architecture projects.

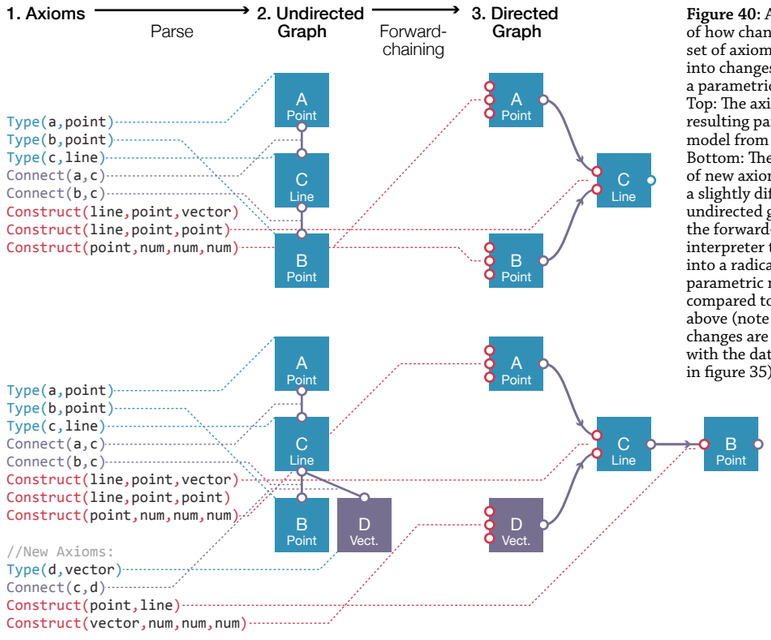


Figure 40: An example of how changes to a set of axioms translate into changes within a parametric model. Top: The axioms and resulting parametric model from figure 36. Bottom: The inclusion of new axioms creates a slightly different undirected graph which the forward-chaining interpreter transforms into a radically different parametric model when compared to the one above (note that these changes are the same as with the dataflow model in figure 35).

5.6 Application to the Sagrada Família

Every vertex was slightly awry on the Sagrada Família's fronton model. This introduced curves to faces that should have been planar, pulled geometry out of line with important axes, and unsettled the proportions of the model (fig. 29). In March 2010, I developed a set of parametric models to realign the vertices of the distorted model. I built these models from Melbourne, Australia and was guided by discussions with Mark Burry and the Sagrada Família design office based in Barcelona, Spain.

To understand the impact of a parametric model's programming paradigm, I straightened the frontons twice: once with a dataflow language, and once with a logic programming language. In order to do so, the geometry of the frontons was first converted into a parametric model in each of the respective programming paradigms. Once converted, parametric relationships were introduced to realign the model. These relations ensured that polygons were regular and that certain groups of vertices were planar, symmetrical, proportioned, and on an axis. The parametric model was then flexed so the new geometry matched the original model as closely as possible.[7] The specific process of using dataflow programming and logic programming was as follows.

Applying Dataflow

The dataflow parametric model contained approximately a thousand geometric operations that straightened the one hundred and eleven vertices in the original fronton model. This parametric model was as large as the largest models analysed in chapter 4.3.[8] Due to the anticipated size of the model, I elected to generate the fronton parametric model in Digital Project. The geometry was imported into Digital Project where I introduced parametric relationships to fix the original model's distortions.

7 Closeness in this case was measured as both the median distance model vertices moved and the maximum distance model vertices moved.

8 Figure 27 & 28 show how the fronton model is only a small component of the Sagrada Família overall, and a fairly geometrically simple component at that. To have such a large and detailed architecture project modelled parametrically is quite unusual. And with parametric models being employed on the Sagrada Família for almost two decades, I would venture to say that in aggregate the models are likely to be the most extensive and most complex parametric models ever used in an architecture project.

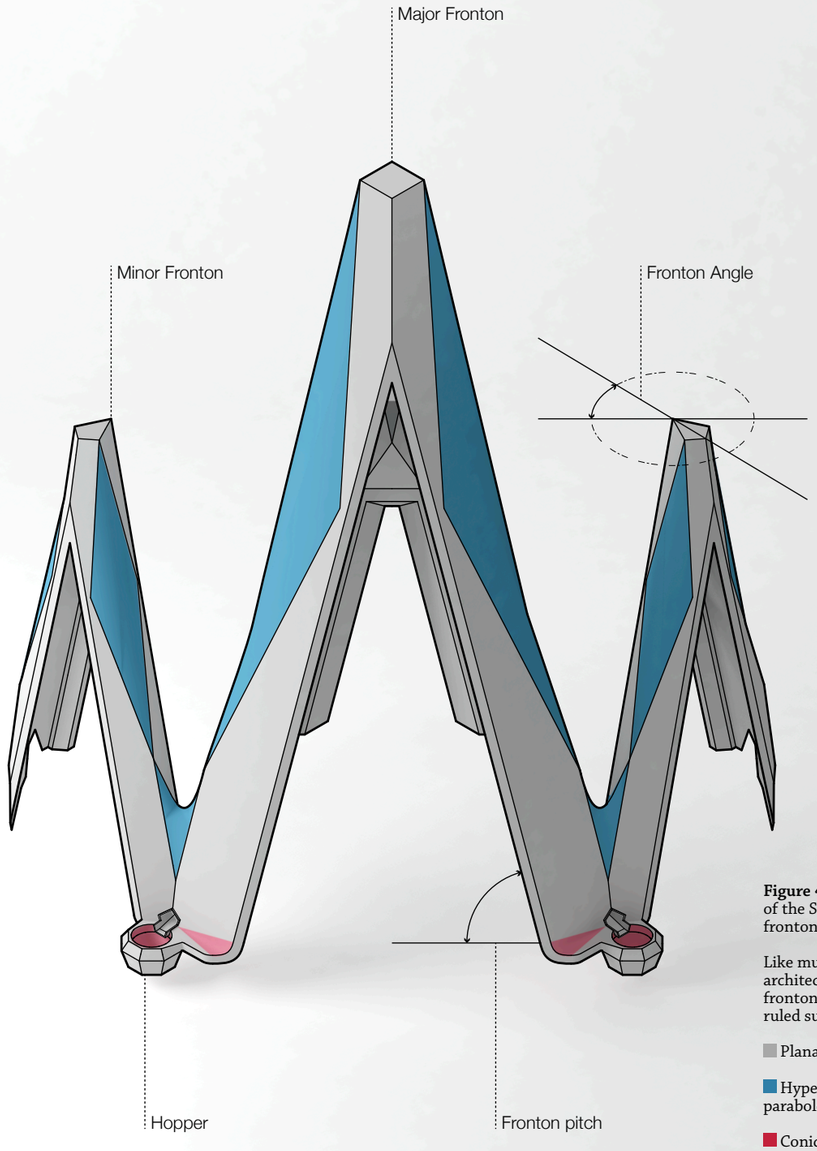


Figure 41: The key parts of the Sagrada Família's frontons.

Like much of Gaudí's architecture, the frontons consist of ruled surfaces:

Planar surfaces.

Hyperbolic paraboloids.

Conic sections.

The parameters of the Digital Project model were managed in an Excel spreadsheet. By changing values in the spreadsheet I could move the new refined frontons as close as possible to the original geometry. The best set of parameters I could find reduced the median difference between the two models to 13mm. I then further refined these values using a genetic algorithm, which narrowed the median distance to 6mm. Since it was only the model's parameters being changed, all the parametric relationships in the new geometry were maintained during this process.

Applying Logic Programming

The frontons were also straightened with the logic programming technique I described earlier. The logic programming environment took a series of axioms describing the frontons' geometric relationships and then derived a parametric model to satisfy these relationships. In total, approximately six hundred axioms were required to generate the parametric model of the frontons. Almost five hundred axioms were automatically generated. These included most of the geometric axioms, which could be found by iterating through all the points, lines, and planes in the original model to produce axioms like: `type(p_127,point)` and `type(plane_3,plane)`. Most of the connection axioms were found using a similar method whereby vertices from lines or planes that coincided with a point were said to be connected, which produced axioms like: `connect(p_127,plane_3)`. The remaining one hundred axioms in the logic programming model were generated manually. These included the construction axioms, as well as new geometric axioms to define important planes, axes, polygons, and vectors. From these axioms the logic programming interpreter generated a parametric model. The model's parameters were refined using a genetic algorithm, which reduced the median distance between the new and old model to 6mm – a comparable result to the dataflow model.[9]

9 The genetic algorithm initially proved ineffective on the parametric models produced by logic programming since the logic programming interpreter initialised all values to zero, which caused the optimisation to begin thousands of millimetres from its target. In pulling the geometry towards the target, the genetic algorithm had a tendency to get stuck on local optima. To overcome this problem, the values of the model were initialised using a hill climbing algorithm. This got the geometry into approximately the right place before the final refinement with the genetic algorithm.

5.7 Analysis of Programming Paradigms

Method

Straightening the Sagrada Família's frontons with a logic programming paradigm and a dataflow paradigm presents the opportunity to observe how the programming paradigms affect the respective parametric models. The following observations draw upon the research instruments discussed in chapter 4. Of particular interest is how the programming paradigm impacts the model's construction time as well as the relative modification time and extendability. In addition, the latency between changes and the verification of model correctness are important differentiators between the two paradigms.

Construction Time

Constructing the first version of the dataflow parametric model in Digital Project and Excel took approximately twenty-three hours. This time does not include the time taken to convert the original geometry into Digital Project or the subsequent time spent modifying the first version of the parametric model. Much of the twenty-three hours was consumed selecting the appropriate parametric relationships, working out how to apply the relationships in a hierarchy of parent-child connections, and verifying the relationships generated the expected geometry. Working out the correct parent-child connections was deceptively difficult since connections often had flow-on implications for the surrounding geometry. These challenges were largely avoided with logic programming by generating the majority of the axioms automatically from the existing geometry and by using the logic programming interpreter to infer the hierarchy of relationships these connections imply. Accordingly, constructing the first version of the logic programming parametric model took approximately five hours.

In this case using a logic programming paradigm was four to five times faster than using a dataflow paradigm. The time difference is largely attributable to the automatic extraction of axioms and subsequent

inference of the model's hierarchy with the logic programming interpreter. This was only possible because there was a pre-existing geometric model of the frontons, which is an unusual circumstance for most architecture projects. The difference in construction time cannot therefore be expected on other architecture projects, particularly ones without a pre-existing geometric model. Nevertheless, the variance in construction time demonstrates that programming paradigms can significantly affect projects, although these affects are dependent upon the circumstances of the project. An appropriate circumstance for logic programming seems to be when a pre-existing explicit geometric model needs to be converted into a parametric model.

Modification Time and Extendability

Modification time is a quantitative measure of how long a change takes to make while *extendability* is a qualitative assessment of the ease with which a program adapts to change (Meyer 1997, 6-7). In straightening the frontons there were two primary changes asked for, both of which reinforce the precision required in the project:

1. **Minor Fronton Angle:** On the original distorted model the minor fronton axis angle was 43.875 degrees (fig. 41). I initially 'corrected' this to 45.000 degrees, which caused the model to move significantly. The design team asked the angle be changed back to 43.875 degrees before subsequently deciding that 43.904 degrees was most in keeping with the geometry of the Sagrada Família's central tower. When I built the dataflow model I was uncertain of the fronton angle so I included a parameter to control it. This parameter permitted these changes to be made almost instantaneously. On the logic programming model the changes could be accommodated by adding a new axiom to define the vector of the plane linked to the centre of the minor fronton. This process took slightly longer than in the dataflow language, but still less than fifteen minutes.

2. **Minor Fronton Pitch:** In one of the final iterations it was discovered that the pitch of the major and minor frontons was slightly different. I had come across the anomaly previously but assumed it was a rounding error since the deviation was less than 0.02 degrees. Over a ten metre

span this slight abnormality resulted in a 2mm error, which was outside the tolerance of the seven-axis robot milling the frontons. The solution was to redefine the pitch of the minor fronton. In the dataflow model this change had few flow-on consequences and therefore took less than thirty minutes to implement. The logic programming language unfortunately lacked the vocabulary to express the pitch change. I spent two hours adding a new construction axiom to the logic programming vocabulary that permitted a vector to be mirrored through a plane. Once this axiom was added, it took approximately fifteen minutes to change the pitch of the minor fronton by mirroring the pitch of the major fronton.

On this project both programming paradigms were extendable enough to accommodate both of the primary changes. The changes were somewhat unusual in that they did not affect the topology of the project (an act many authors I discussed in chapter 2.3 found disruptive). The changes instead focused on the precision of the model and the way the geometry was related. In making these changes the dataflow language offered slightly faster modification times: in the first case because I anticipated the change by including a parameter for it, and in the second case because logic programming was slowed by limitations in its vocabulary. This success does not confirm the agility of dataflow programming as much as it confirms the importance of the designer's intuition in setting up a model's hierarchy, and the importance of what Meyer (1997, 12-13) calls the modelling environment's functionality (having the right vocabulary to express an idea or change).

In retrospect the changes to the frontons seem relatively minor for the effort expended on them. However, the magnitude of these changes comes from the model extendability rather than the brief. Had I been using a non-parametric model, or had I created an inflexible parametric model, both of these changes would have involved deleting over half of the model's geometry and starting again. The minor changes would have been serious problems. Thus, while the attention to detail on the Sagrada Família may seem pedantic, the level of design consideration is only afforded by maintaining flexible representations.

Latency

Latency measures the delay in seeing the geometry change after changing a model's parameters. With the dataflow model, modifications to the Excel values would propagate out into the geometry produced by Digital Project within a few seconds, which was not quite real-time but near enough for this project. Conversely, the logic programming model had a pronounced latency between editing the axioms and seeing the resulting geometry. Minutes would elapse while the interpreter worked to derive the parametric model after axiom changes.[10] Since the derived parametric model did not always have parameters in intuitive places, often the only way to change numerical values (like the angle of the minor fronton) was to change the axioms and then wait as the interpreter derived a new parametric model. This latency made changing the logic programming model a more involved and less intuitive process than with the dataflow model.

Correctness

Correctness describes whether a program does what is expected (Meyer 1997, 4-5). For both programming paradigms it was difficult to verify the models were doing what was expected. In the dataflow language the quantity of relationships obfuscated the flow of data, which made it hard to work out what the model should have been doing. On three occasions this led to the wrong geometric operation being applied. These errors were not apparent by looking at the dataflow model or by visually inspecting the geometry, and they were only caught by the project architects manually measuring the model. Logic programming was equally difficult to verify because it was not always apparent how the geometry derived from the axioms. Often I would end up adding axioms one-by-one to understand their impact on the final geometry. In the end, both programming paradigms produced parametric models that did what was expected but, in a project where the geometric changes were subtle and the relationships numerous, often the only way to verify correctness was to examine what the model did rather than understand how the model did it.

10 The latency increases with model size although it may be influenced by other factors. These include the interpreter's efficiency and the way the axioms define the problem. The results discussed should not be interpreted as evidence that logic programming suffers from general latency issues, rather the results should be read with the caveat that they are particular to the project's specific logic programming implementation and to the specific circumstances of this project.

5.8 Conclusion

Van Roy and Haridi (2004, xx) write that when it comes to programming paradigms "more is not better (or worse), just different." Yet for architects applying parametric models to projects like the Sagrada Família, they have always had less than more when it comes to programming paradigms. Architects building parametric models are presently confined to either declarative dataflow paradigms in visual programming languages, or procedural imperative paradigms in textual programming languages. The research I have presented demonstrates that programming paradigms influence – at the very least – the parametric model's construction time, modification time, latency, and extendability. Since programming paradigms cannot normally be switched without rebuilding a model, selecting an appropriate programming paradigm for the context of a project is a critical initial decision. This is a decision that has an evident affect on many aspects of a parametric model's flexibility yet, unfortunately, it is a decision largely unavailable to architects; they often cannot choose more, less, or different – just: dataflow or procedural. Van Roy and Haridi's discussions of programming languages are interesting, not just because Van Roy sees the Sagrada Família as a metaphorical signifier of them, but because these discussions of programming paradigms signify how parametric models, like those used on the Sagrada Família, can be tuned to privilege different types of design changes.

Perhaps the most intriguing part of this case study is the disjunction between the theoretical advantages of logic programming and the realised advantages of logic programming. I began this chapter by talking about the challenges of modifying the hierarchy of relationships in a dataflow language and I postulated that the time associated with these modifications could theoretically be reduced if the computer – rather than the designer – organised the parametric model's hierarchy. In a series of diagrams I illustrated how a logic programming language would allow a designer to specify directionless connections that are automatically organised by a logic programming interpreter, thereby reducing the theoretical modification time. Yet in reality the opposite happened: when used to generate the parametric model of the Sagrada Família's frontons, logic programming actually lengthened the modification time. This is in large part because the interpreter often took a long time to generate a new instance of the model – a detail easily overlooked when discussing logic programming

Figure 42: The frontons under construction at The Sagrada Família in September 2012.

theoretically, but one that can possess significant importance when a designer has to wait five minutes for the logic programming interpreter to work through the six hundred axioms in order to realise a single modification to the frontons. While these results are highly dependent upon circumstance – and although the Sagrada Família's size, detail, and precision makes it a rare circumstance – these results do underscore the value of practice-based understandings of parametric modelling. This disjunction between theory and practice when talking about programming paradigms is perhaps the reason why logic programming failed to live up to the theoretical expectations that had been expressed by authors who had never actually used logic programming in practice – examples include, Mitchell (1990) in *The Logic of Architecture* and Swinson (1982) in *Logic Programming: A computing tool for the architect of the future*.

While logic programming did not live up to Mitchell or Swinson's hopes, and while logic programming induced a far greater latency than I expected in theory, this is not to say logic programming is without merit. Logic programming in architecture, as in software engineering, appears to excel at reasoning about relationships. In this capacity, it seems particularly suited to extracting relationships from pre-existing geometry in order to derive a parametric model; the reverse of the typical parametric modelling process. In the fronton case study, with a pre-existing geometric model of the frontons, this led to significantly reduced construction times when compared to the dataflow model. However, both methods produced large and intricate models that were hard to verify as being correct – an issue of understandability addressed in the following chapter.

6 Case B: Structured Programming

Project: Designing Dermoid.

Location: Royal Danish Academy of Fine Arts, Copenhagen, Denmark.

Project participants from SIAL: Mark Burry, Jane Burry, Daniel Davis, Alexander Peña de Leon.

Project participants from CITA: Mette Thomsen, Martin Tamke, Phil Ayres, Anders Deleuran, Aron Fidjeland, Stig Nielsen, Morten Winter, Tore Banke, Jacob Riiber; Workshop 3, Material Behaviour, Department 2, EK2, fourth year (November 2010); Workshop 4, Paths to Production, Department 2, third year, (January 2011).

Related publications:

Davis, Daniel, Jane Burry, and Mark Burry. 2011. "Understanding Visual Scripts: Improving collaboration through modular programming." *International Journal of Architectural Computing* 9 (4): 361-376.

Davis, Daniel, Jane Burry, and Mark Burry. 2011. "Untangling Parametric Schemata: Enhancing Collaboration through Modular Programming." In *Designing Together: Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures*, edited by Pierre Leclercq, Ann Heylighen, and Geneviève Martin, 55-68. Liège: Les Éditions de l'Université de Liège.

Figure 43: Digital and physical models intermixed at the June 2010 Dermoid Workshop. From left to right: Martin Tamke, jacob Riiber, Morten Winter, Jane Burry (hidden), Mark Burry, Alexander Peña de Leon, Phil Ayres, Mette Thomsen.

6.1 Introduction

In June 2010 I found myself biking to the edge of Copenhagen, out past the Royal Danish Academy of Fine Art, and into a secluded concrete studio. The studio was filled with full-scale wooden prototypes, with laptops connected to various international power adaptors, and with researchers from CITA[1] and SIAL[2]. The researchers were all debating a deceptively simple problem: how can we fashion a doubly curved pavilion from a wooden reciprocal frame. It is a question that would occupy a dozen researchers, including myself, for over a year; a question that would eventually led to the construction of the first Dermoid pavilion in March 2011.

1 Center for Information Technology and Architecture at the Royal Danish Academy of Fine Arts, Copenhagen
2 Spatial Information Architecture Laboratory at RMIT University, Melbourne

Figure 44: Detail of Dermoid installed at the 1:1 Research By Design exhibition, March 2011, Royal Danish Academy of Fine Art, Copenhagen.

Figure 45: Dermoid installed at the 1:1 Research By Design exhibition, March 2011, Royal Danish Academy of Fine Art, Copenhagen.

There are numerous reasons why Dermoid was so difficult. One source of difficulty was the unknowns in the brief: we did not know the shape of the pavilion's doubly curved surface, or even where the pavilion would be built; and at the time no one could calculate the structural performance of a reciprocal frame, especially one constructed from a heterogeneous material like wood. There were also many known difficulties in the brief: uniformly patterning a doubly curved surface is notoriously hard, and the circular relationships of a reciprocal frame do not lend themselves to parametric modelling. Further adding to the difficulty, the project involved a diverse team situated at opposite ends of the earth. In short, it was a project destined to challenge even the most skilled designers, the ideal project to observe the inflexibility of parametric models.

While Dermoid embodies many noteworthy innovations, in this case study I want to discuss specifically the development of Dermoid's parametric models. The models have many authors since the researchers working on Dermoid were all experienced in parametric modelling, many of them world experts. The range of contributors meant that there was rarely a single "keeper of the geometry" – a name Yanni Loukissas (2009) gives to the person on a project who inevitably becomes solely responsible for the upkeep of the model. I assumed this role briefly as I prepared Dermoid's parametric models for a workshop held in November 2010 at the Royal Danish Academy of Fine Arts. During this period I experimented with changing the structure of the models based on organisational techniques used by software engineers (identified in chapter 3.2). In this chapter I consider the impact of these changes using a combination of thinking-aloud interviews and observations of subsequent model development. I will begin by discussing the historic motivation that led software engineers to structure their code, and benefits they observed from doing so.

6.2 Structured Programming

In March 1968, Edsger Dijkstra (1968) wrote a letter to the Association for Computing Machinery entitled *Go To Statement Considered Harmful*. At the time, the GOTO statement was the primary mechanism of controlling a computer program's sequence of execution (the GOTO statement allows a program to skip ahead or jump backwards in a chain of programming commands). Dijkstra (1968, 148) argued that the intertwined jumps programmers were producing with GOTO statements were "too much an invitation to make a mess of one's program." Building on the work of Böhm and Jacopini (1966), Dijkstra proposed reducing the mess with simple structural commands such as if-then-else, and while-repeat-until. Although these structures now underlie all modern programming languages, they were not an obvious development in 1968. Many worried that structure would interrupt the "art" of programming (Summit 1996, 284), and that code would be even more difficult to understand when obscured by structure. Dijkstra (1968, 148) agreed and cautioned, "the resulting flow diagram cannot be expected to be more transparent than the original one." Nevertheless, when scientists assembled at NATO a few months later in 1968 to discuss the impending software crisis – with Dijkstra in attendance – many of their conversations made reference to code structure (Naur and Randell 1968).[3]

While there was no single cure to the software crisis, structure is now recognised as an important remedy for taming what Bertrand Meyer (1997, 678) calls the "unmistakable 'spaghetti bowl' look" of tangled GOTO statements that undoubtedly contributed to parts of the crisis. There are many types of structure but Böhm and Jacopini's (1966) original proof (referred to by Dijkstra) uses only three, which they represent by the symbols ∏, Ω, and Δ (fig. 46). Böhm and Jacopini (1966) showed how these three structures could be combined, without the GOTO statement, to create Turing complete programs. The implication of their proof is

3 At the meeting unstructured code was never singled out as one of the a causes of the software crisis. In fact, none of the attendees in the meeting minutes (Naur and Randell 1968) make reference to unstructured programming or the GOTO statement. They do however often talk about code structure and code modules. Dijkstra also presented a paper entitled *Complexity Controlled by Hierarchical Ordering of Function and Variability* where he describes grouping code into layers that are restricted so they can only communicate with layers above them. While there are structural principles to this idea, it is a different type of structure to the one Böhm and Jacopini (1966) discussed and that Dijkstra (1968) referred to in *Go To Statement Considered Harmful*. In essence, structure was an idea that was gaining traction around the time of the NATO conference, but one that was still in the early stages of taking shape.

Δ	Sequence: Executing a subprogram in order.
Ω	Iteration: Executing a subprogram until a condition is reached.
∏	Selection: Executing a subprogram based on a condition.

Image removed to comply with RMIT University copyright regulation. Please download unedited thesis from: http://www.danieldavis.com/thesis/

Figure 46: The three structures that Böhm and Jacopini (1966) proved could be combined to create a Turing machine.

that any unstructured program employing the GOTO statement can be rewritten without the GOTO statement by decomposing the program into a structure of subprograms that are linked using ∏, Ω, and Δ. Doing so eliminates the danger of stray GOTO statements jumping into unexpected locations. However, it took a lot more than a letter from Dijkstra for this proof to filter down into practice.[4]

Modules

The subprograms employed by Böhm and Jacopini have many synonyms in contemporary programming: methods, functions, procedures, and modules. Each term signifies the same general idea with a slightly different overtone. I have chosen to use the word *module* because of the connotations with standardisation, reuse, self-containment, and assembly (themes I will explore further in this chapter). A module is defined by Wong and Sharp (1992, 43) as "a sequence of program instructions bounded by an entry and exit point" that perform "one problem-related task" (these principles are applied to a module in Grasshopper in figure 47). If employed

4 While Böhm and Jacopini (1966) had shown that it was theoretically possible to write programs without the GOTO statement, this was not possible in practice until programming languages could accommodate Böhm and Jacopini's three structures: sequence, iteration, and selection. Even after the development of these languages, programmers who were comfortable using the GOTO statement still used it. And nineteen years after Dijkstra's (1968) original ACM letter – *Go To Statement Considered Harmful* – people were still writing rebuttals in the letters to the ACM like Frank Rubin's (1987) *"GOTO Considered Harmful" Considered Harmful*.

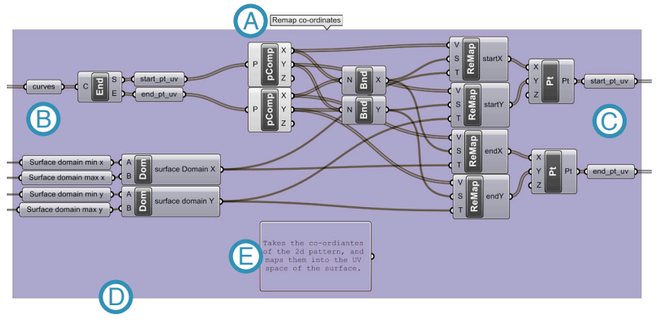


Figure 47: A typical module in Grasshopper. The grey boxes are operations (themselves small modules) that have been linked together to form a larger module. More recent versions of Grasshopper have native support for modules (which are called clusters in Grasshopper) however at the time of my research this version of Grasshopper had not been released.

A: The name of the module.

B: The inputs – the only place data enters the module.

C: The outputs – the only place data leaves the module.

D: The operations of the module are encapsulated so that they can only be invoked by passing data through the module's inputs.

E: A description of what the module does – a module does one problem-related task.

successfully, modules have five principle benefits according to Bertrand Meyer (1997, 40-46):

1. **Decomposition:** A complicated problem can be decomposed into a series of simpler sub-problems each contained within a module. Decomposing problems like this may make them easier to approach and may make it easier for teams to work together since each team member can work on a separate sub-problem independently.

2. **Composition:** If modules are adequately autonomous they can be recombined to form new programs (a composition). This enables the knowledge within each module (of how to address a sub-problem) to be shared and reused beyond its original context.

3. **Understandability:** If a module is fully self-contained, a programmer should be able to understand it without needing to decipher the overall program. Conversely, a programmer should be able to understand the overall program without seeing the implementation details of each individual module. Dijkstra (1968, 148) worried this would lead to less transparency but most have since argued that abstraction helps understandability. For instance, Thomas McCabe (1976, 317) has posited that modularisation improves understandability since it reduces the cyclomatic complexity, making it "one way in which program complexity can be controlled." Meyer (1997, 54) points out that modularisation aids a programmer's comprehension of the code through the names given to inputs, outputs, and the module itself.

4. **Continuity:** A program has continuity when changes can be made without triggering cascades of other changes. In a program without continuity, changing one module will affect all the dependent modules,

setting off a chain-reaction as all the dependent modules are changed to accommodate the original change and so on. Continuity has much to do with how a program's structure is decomposed. David Parnas (1972, 1058) suggests that projects should be broken around "difficult design decisions or design decisions which are likely to change" so that each anticipated change is contained within a module in such a way that it does not impact the other modules.

5. **Protection:** Each module can be individually tested and debugged to ensure it works correctly. But if something does go wrong within a module, the module can contain the error and thwart its propagation throughout the program (protecting the rest of the modules from the error).

The benefits of modularisation are so pervasive that some modern programming languages, like C# and Java, make it impossible to write code not contained within some sort of module. Java even stopped supporting the GOTO statement, and some of the more recently invented languages – like Python and Ruby – have never supported the GOTO statement.[5] In its place are screeds of structural constructs, from switch-case, to try-catch, to polymorphic objects. These structures, like Böhm and Jacopini's original three, offer programmers various ways to decompose and recompose programs from smaller, self-contained chunks. Debates continue about how best to wield structure in order to increase understandability and reduce complexity, whilst improving continuity and protection. These debates fill entire sections of libraries and occupy the *Software Design* [2.2] section of the *Software Engineering Body of Knowledge Version 1.0* (Hilburn et al. 1999, 20). Yet despite the pervasive benefits of modularisation, architects creating parametric models in visual programming languages still tend to create unstructured models, as I will show in the following section.

5 Neither Python nor Ruby support the GOTO statement by default but it can be turned on in Ruby 1.9 by compiling with the flag SUPPORT_JOKE and it can be added to Python by importing a library Richie Hindle created as an April fools joke in 2004 (http://entrian.com/goto/). The jesting about adding GOTO to Ruby and Python speaks volumes of their relationship with the GOTO statement.

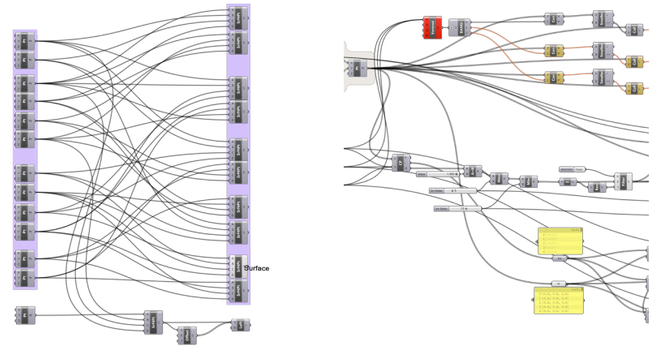


Figure 48: Examples of *spaghetti* forming in two unstructured Grasshopper models. Neither model gives any hint (through naming or otherwise) as to what the crisscrossed connections do and it is impossible deduce simply from inspection.

6.3 Architects Structuring Visual Programs

Unlike the programming languages in the 1960s, which were unstructured simply because the syntax for structure had not been invented, all the major visual programming languages used by architects have some basic structural constructs. In particular, they all support modularisation. In the lexicon of the various software, modules have come to be known as *features* in Bentley's GenerativeComponents, *digital assets* in Sidefx's Houdini, and *clusters* in McNeel's Grasshopper. But even though these modular constructs exist, architects tend not to use them. In chapter 4.3's sample of 2002 Grasshopper models, 97.5% of the models did not employ modules.[6] Moreover, 48% of models had no modules, no groups, no explanation of what they did, and no naming of parameters: by even the most generous of definitions these models were completely unstructured.[7] In addition to being unstructured, the models generally have a high cyclomatic complexity (see chap. 4.3) and possess what Meyer (1997, 678) calls the "unmistakable 'spaghetti bowl' look" of interwoven relationships and long chain dependencies (fig. 48). In many ways these tangled visual programs parallel

6 1553 of the sampled models were created in a version of Grasshopper that supported clusters (either below version 0.6.12 or above version 0.8.0) and of these models only 39 contained at least one cluster.

7 In the sample of 2002 Grasshopper models, 36% of the models contained at least one piece of text that explained what part of the model did; 30% of the models used one or more groups; 19% of the models had at least one node that named a branch of data; 2.5% of the models had clusters; and 48% had none of the above. This does not mean the other 52% are entirely structured; even though a model is structured by groups, and explanations, and names, their presence does not guarantee that the model is structured (for example, the unstructured models in figure 48 are part of the 52% since they both use groups). The percentage of unstructured models therefore falls somewhere between 48% and 97.5% depending on the definition of structure, but I would assume most definitions would conclude that at least 90% of the sampled models are unstructured.

the knots of GOTO statements that characterised the programs of the 1960s, with seemingly similar consequences in terms of understandability. It remains unknown precisely why architects are creating models that are seemingly so messy, complicated, and unstructured. Two possible explanations are that the Grasshopper implementation of modules is somehow flawed, or that architects lack the knowledge required to utilise the modules properly.[8]

8 A third explanation has been put forward by others I have spoken to: architects are under too much pressure to bother structuring their models. As Woodbury (2010, 9) puts it, architects quickly "find, skim, test and modify code for the task at hand" and then move onto the next one leaving "abstraction, generality and reuse mostly for 'real programmers'." I find this explanation unconvincing because it ignores the fact that many software engineers are also working under a lot of pressure. If software engineers and architects both experience pressure, then pressure alone does not explain why one group so studiously structures their programs while the other group almost never does.

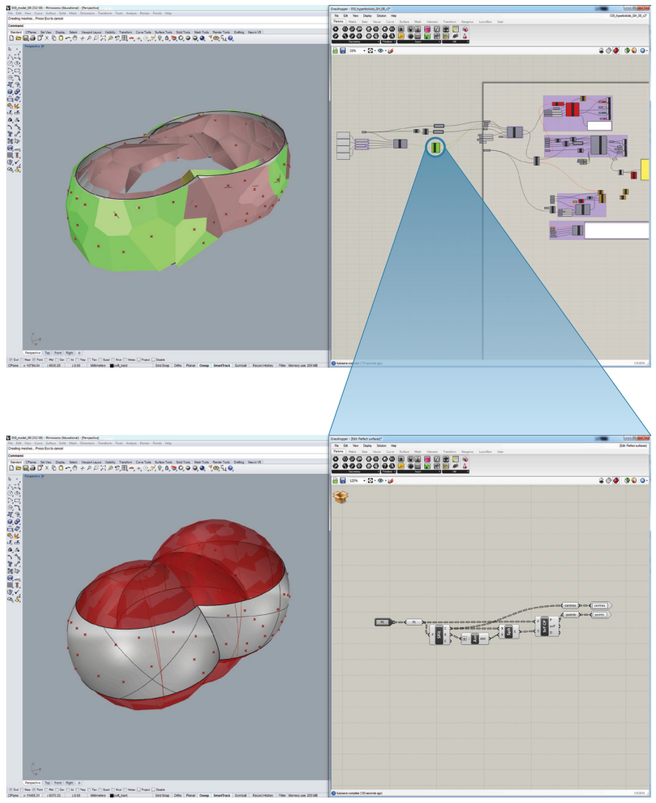


Figure 49: A cluster in Grasshopper (a model used in chapter 7 for the FabPod). Top: The full parametric model with the cluster in its most abstract form. Bottom: Opening the cluster to reveal the operations it encapsulates, however, opening the cluster also hides the rest of the model, which impedes the model's visibility and juxtaposability.

Implementation of Clusters

The low use of modules may be in part an artefact of Grasshopper's cluster implementation. Clusters were a feature present in early versions of Grasshopper that was later removed in version 0.6.12 and subsequently reintroduced in version 0.8.0. The inconsistent presence and function of clusters undoubtedly makes some users untrusting of them.

Perhaps more significantly, however, the way clusters are currently implemented in Grasshopper may actually impede the understandability of the model.[9] As Dijkstra (1968, 148) warned, structure can make the resulting program less "transparent than the original one." While the abstraction brought about by less transparency may be beneficial in a textual language, in a visual language structural abstractions can hinder access to code according to Green and Petre (1996, 164). Their widely cited research on the usability of visual programming languages indicates that the understandability of a program is dependent upon *visibility* (how readily parts of the code can be seen) and *juxtaposability* (the ability to see two portions of code side-by-side) (Green and Petre 1996, 162-164). Clusters in Grasshopper constrain visibility by limiting the view to one particular level of abstraction at a time (fig. 49). Juxtaposability is currently impossible in Grasshopper since two levels of abstraction cannot be seen at the same time, or side-by-side. Furthermore, cluster reusability is impeded since cluster changes do not propagate through related instances of reused clusters. Owing to these limitations, the clusters in Grasshopper are more suited to packaging finalised code rather than supporting the decomposition and composition of an evolving program (the way structure is typically used in textual programs). This may be one reason for low cluster use in Grasshopper.

9 At the time of writing (late 2012) Grasshopper is still under development. This description of clusters in Grasshopper helps explain why clusters and structure were not in the models I sampled, but it may not apply to models created in future versions of Grasshopper since the cluster implementation is likely to change.

Structure and Education

Another possible factor leading to low cluster use has to do with the education of architects. Designers are generally not taught about parametric modelling as much as they are taught to use parametric modelling software.[10] Woodbury (2010, 8) observes that most manuals and tutorials teach students by "providing lists of commands or detailed, keystroke-by-keystroke instructions to achieve a particular task." For example, a student learning to use Grasshopper may start with the *Grasshopper Primer* (A. Payne and Issa, 2009). On page twenty-seven they learn how selecting seven items from the menu and linking them together produces a spiral through points, which is a lesson that is not substantively different to learning how selecting two items from the menu in the non-parametric software, Rhino, will also produce a spiral through points. This pedagogy continues throughout the *Grasshopper Primer* and in other Grasshopper introductions like Zubin Khabazi 's (2010) *Generative Algorithms using Grasshopper* as well as in the teaching material for other parametric modelling software like Bentley Systems' (2008) *GenerativeComponents V8i Essentials* and Side Effects Software's (2012) *Houdini User Guide*. Students using these various guides are primarily taught the particular sequence of interface actions to make a tool that produces a particular geometric outcome, almost always without being taught the accompanying abstract concepts like program structure.

This parametric modelling pedagogy contrasts sharply with how programmers are taught. In chapter 3.2 I showed how the basic skill of programming (knowing the particular sequence of interface actions to produce a particular outcome) forms only a small part of the *Software Engineering Body of Knowledge Version 1.0* (Hilburn et al. 1999). Programming is therefore only a small part of what entry level programmers are expected to know. Even resources designed to teach the basic skill of programming cannot help but discuss more abstract structural concepts – for instance, the fifth, sixth, and seventh chapters of *Beginning Python* (J. Payne 2010) respectively cover the following: creating subprograms and functions; creating classes and objects; and structurally organising programs.[11] Structure is such an intrinsic part of programming that it is mandatory in some languages, like Java and C#, a concept reinforced in the practice of programming and fundamental to the education of programmers.

For architects, the most comprehensive analysis of structuring parametric models comes from Woodbury, Aish, and Kilian's (2007) paper, *Some Patterns for Parametric Modeling*, which was later republished as a sizeable part of Woodbury's (2011) *Elements of Parametric Design*. The paper riffs on the seminal software engineering book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides[12] (1995), although each has a slightly different emphasis: *Design Patterns* focuses on methods of structuring code to address problems with the code itself (such as reusability, understandability, and extendability), whereas *Some Patterns for Parametric modeling* presents patterns that solve problems specific to architecture (such as ordering points, projecting geometry, and selecting objects). This makes *Some Patterns for Parametric Modeling* more like a recipe book of useful modules than a *Design Patterns*-esque guide for structuring programs.

One pattern from *Some Patterns for Parametric Modeling* does address problems with the understandability of code itself. The *Clear Names* pattern advocates always naming objects with "clear, meaningful, short and memorable names" (Woodbury 2010, 190). This is a relatively easy pattern to follow in Grasshopper since the names of parameters can be quickly changed by clicking on them. Yet neither of the training manuals provided on the official Grasshopper website teach architects the clear names pattern. The only reference in *Generative Algorithms Using Grasshopper* comes from a caption that mentions "I renamed components to point A/B/C by the first option of their context menu to recognize them easier [sic] in canvas" (Khabazi 2010, 11). Similarly, the only reference in the *Grasshopper Primer* is half a sentence mentioning that designers can "change the name to something more descriptive" (A. Payne and Issa 2009, 10), without explaining how or why they should. Not surprisingly, 81% of the Grasshopper models I sampled contained no uniquely named parameters. This absence of basic modifications that improve the

10 This teaching method has been advanced since at least 1989 when Alexander Asanowicz argued at eCAADe "we should teach how to use personal computer programs and not programming."

11 While I have chosen *Beginning Python* to illustrate this point, the same is true of almost any book on programming.

12 *Design Patterns* is in turn based upon the work of Christopher Alexander.

understandability of models may be a symptom of how architects are taught to model. While programmers learn about structure in basic books like *Beginning Python* and in dedicated books like *Design Patterns*, even simple concepts like naming parameters cannot be found in the educational material given to architects. This may be one reason that more advanced structural techniques (like modules) are so infrequently used by architects.

To Understand Visual Programs Better

The benefits of structured programming are undebatable for contemporary software engineers; it is something all programmers do, something some languages mandate, something covered in even basic introductions to programming. Despite the strong evidence in software engineering that structure is beneficial, we know very little about how the structure of parametric models affects the practice of architecture. We do know that architects tend not to structure their models, with two possible factors being both the education of architects and the way modules are implemented in parametric software. In this case study I consider what happens if these two impediments are removed and an architect structures their model. In particular I examine whether overcoming such impediments would be a worthwhile pursuit for architects. I have spread these considerations over a series of three experiments related to structuring the parametric models of the Dermoid pavilion:

1. Evaluating the understandability of structured programs through thinking-aloud interviews [6.4].
2. Analysing Dermoid's modular model structure and how this affected the project development [6.5].
3. Consideration of how parts of Dermoid can be recomposed and shared with the internet [6.6].

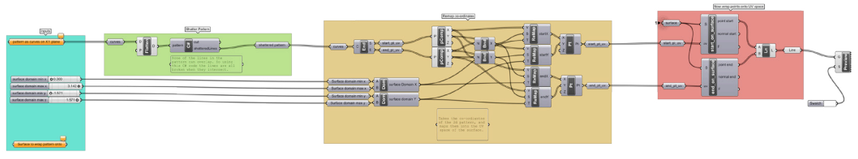
6.4 Understandability of Visual Programs in Architecture

To discern whether structuring a parametric model impacts an architect's comprehension of the model, I conducted an experiment whereby architecture students were shown a series of structured and unstructured visual programs. Using a thinking-aloud interview technique I established how legible the students found models with and without structure, thereby articulating what architects may or may not be missing when they create visual programs devoid of structure.

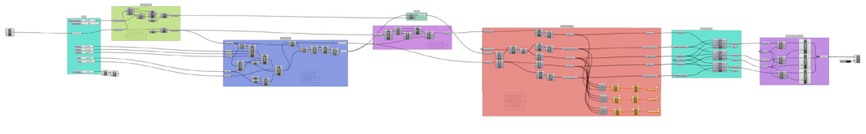
Method

Thinking-aloud interviews are a type of protocol analysis commonly used in computer usability studies as a means of understanding a user's thought process as they carry out a task (Nielsen 1993, 195-200; Lewis and Rieman 1993, 83-86). Clayton Lewis pioneered the technique while working at IBM, a technique he plainly describes as "you ask your users to perform a test task, but you also ask them to talk to you while they work on it" (Lewis and Rieman 1993, 83). Users are typically asked to discuss the "things they find confusing, and decisions they are making" (Lewis and Rieman 1993, 84). As participants answer these questions they hopefully give the researcher an insight into their experience of performing the tasks; insights that would otherwise be concealed if the researchers only examined the participants actions, or only asked the participants point-blank, *how easy was this task?*

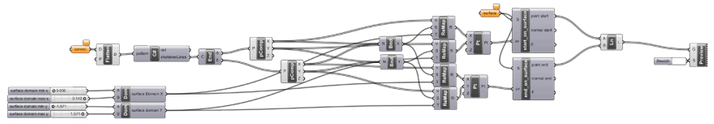
The participants were randomly selected from a class of twenty-five architecture students from the Royal Danish Academy of Fine Art who were attending a weeklong parametric modelling workshop. Four students were selected based on usability expert Jacob Neilson's (1994, 249-56) recommendation to use between three and five participants in thinking-aloud interviews. The selected students each had between one and seven years' experience with computer-aided design, and all had one year's experience using Grasshopper – making them competent users but by no means experts. Each participant was shown three Grasshopper models in a prescribed order (fig. 50). For every model presented, the participant was



Model-A1
Nodes: 41
Structured: Yes
Function: Wraps two-dimensional pattern onto a surface
Equivalent to: Model-C1

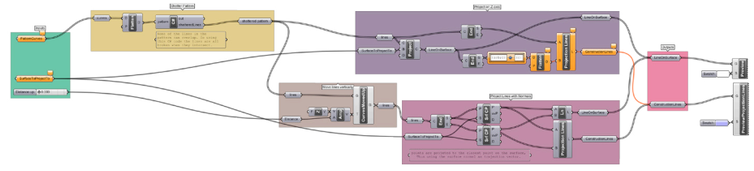


Model-B
Nodes: 120
Structured: Yes
Function: Draws triangles on a hemisphere from an inscribed polyhedron
Equivalent to: n/a

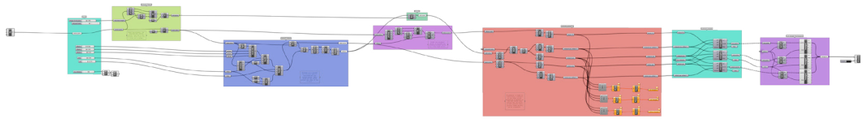


Model-C1
Nodes: 26
Structured: No
Function: Wraps two-dimensional pattern onto a surface
Equivalent to: Model-A1

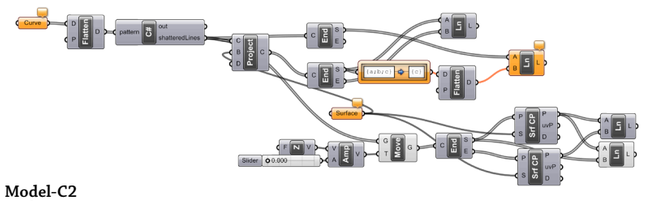
Figure 50: The Grasshopper models shown to the participants. To reduce the bias from one model being uncharacteristically understandable the participants were either shown the three models on this page or the three models on the facing page (selected at random). The first model the participants saw, model-A, was a structured versions of the last model the participants saw, model-C. These models were of an average size (see chap. 4.3) and did a task the participants were generally familiar with (applying two-dimensional patterns to three-dimensional surfaces). To mask the fact that model-A and model-C were equivalent, the participants were shown model-B in between, which was much larger and did a task the participants were unfamiliar with (to ensure the participants spent a long time studying the model and forgetting about the first model). As the experiment was conducted at a time when Grasshopper did not support clusters, the structure was generated through visually separating groups of code around defined entry and exit points, and through clearly naming parameters and groups. Fortuitously this avoids some of the aforementioned issues of visibility and juxtaposability present in Grasshopper's current cluster implementation.



Model-A2
Nodes: 39
Structured: Yes
Function: Projects two-dimensional pattern onto a surface
Equivalent to: Model-C2



Model-B
Nodes: 120
Structured: Yes
Function: Draws triangles on a hemisphere from an inscribed polyhedron
Equivalent to: n/a



Model-C2
Nodes: 20
Structured: No
Function: Projects two-dimensional pattern onto a surface
Equivalent to: Model-A2

set the task of describing how the model's inputs controlled the model's geometry (which was hidden from view) while talking-aloud about their reasoning process. This essentially placed the participants in a role similar to a designer trying to understand a parametric model a colleague had shared with them. The participants were free to explore the model by dragging, zooming, and clicking on screen.

Unbeknownst to the participants, the only difference between the first model they saw (model-A) and the last model they saw (model-C) was the structure of the two models. This allowed me to observe a designer reading a structured model and then reading again the unstructured version of the same model. I was then able to compare how structure affected the understandably of the two models. To mask the similarities of the first and last model, the participants were shown a much larger model (model-B) in between seeing the structured model-A and its unstructured equivalent, model-C. None of the participants realised they had been shown two versions of the same model.

Thinking-Aloud Results

When shown the structured model (model-A) the participants could all describe the model's overall function. They had no problems identifying the inputs or outputs, and half could describe what occurred in each of the model's major stages. When asked about individual nodes, the participants generally understood what each node did but on occasion they would struggle to explain the precise outcome of a particular node within its context.

In contrast, when shown the same model in unstructured from (model-C) all the participants resorted to guessing the model's function (none guessed correctly). A typical comment from Participant-2 was: "It relaxes the lines? That's a guess though, because I am not sure what any of these elements [nodes], I am not sure what any of them do." In reality all the participants knew what each node did; when asked about individual nodes they would be able to say things like "it [the node] makes a line that joins two points." What Participant-2 was struggling with – like all the participants – was assembling this understanding of individual nodes into an understanding of the aggregate behaviour of all the nodes. With no structure to guide

them, the participants often missed important clues like identifying the model's inputs. No participant even realised they were being shown an unstructured version of the model they had seen earlier – all were surprised when told afterwards.

That participants should find structured models more understandable than unstructured models is hardly surprising given the aforementioned practices of software engineers. Yet it is surprising to see how relatively incomprehensible unstructured models – even small ones – are to designers unfamiliar with them. Even the much larger model-B was better understood by the participants than the small and unstructured model-C. Despite model-B's size and fairly obscure function, the participants could all methodically move through the nodes in each module describing them in far better detail than they could with model-C (although their understanding was not as comprehensive as with model-A). While size seems to invite complexity (see chap. 4.3), it seems that structure largely determines a model's legibility.

The structured models had a number of key elements that seemed to guide the participant's comprehension:

- **Names:** Participants regularly referred to node names and module names as they explained the model. This reinforces the *Clear Names* design pattern advocated by Woodbury (2010, 190). While naming nodes is relatively easy in Grasshopper, in the sample of 2002 Grasshopper models, only 19% of the models had one or more nodes that named a branch of data.

- **Positioning:** Participants often overlooked critical input nodes and output nodes in model-C since the unstructured model had all the nodes intermixed. Yet in the structured models (where all the inputs were to the left and all the outputs to the right) the participants could readily identify the inputs and outputs.

- **Explanations:** Some of the modules inside model-A and model-B contained short explanations of what they did. Participants seldom took the time to read these, which indicates a self-documenting model (one with clear names and a clear structure) is preferable to one explained through external documentation.

- **Grouping:** Two participants cited the grouping of nodes, and particularly how they were coloured, as a major aid. As with naming nodes, grouping nodes is relatively easy in Grasshopper, but it is not done by the majority of users (70% of the 2002 sampled models had no groups in them).

Factors in Understandability

There are different theories about how programmers come to understand code (Détienne 2001, 75) but all agree it is fundamentally a mapping exercise between the textual representation and the programmer's internal cognitive representation. While the precise mechanisms of this mapping remain hidden, Green and Petre (1996, 7) observe that "programmers neither write down a program in text order from start to finish, nor work top-down from the highest mental construct to the smallest. They sometimes jump from a high level to a low level or vice versa, and they frequently revise what they have written so far." This jumping between levels corroborates with Meyer's (1997, 40-43) suggestion that structure helps programmers both to *decompose* high level ideas into smaller concepts, and to *compose* smaller parts into larger conglomerates. Yet my research has shown that the vast majority of architects neither compose nor decompose, they instead arrange components at one fixed level of abstraction. Architects presumably have in mind an overall notion of how the model works, but it seems without structure this overall perspective is lost along with the model's legibility to designers who did not create the model. Designers are left to deduce a model's overall behaviour solely through understanding the interaction of the model's parts, which is an inference that none of the participants I observed came close to making.

The key finding of these thinking-aloud interviews is that designers find mapping between unstructured representations and their own internal cognitive representations difficult, if not impossible. Structure does not

just make these mappings easier, it largely determines whether they can be done at all. This is a concerning finding in light of how infrequently architects structure their models. Most designers could introduce structure with a few key alterations, the most effective of which seem to be: clearly naming parameters, grouping nodes together, and providing clearly defined inputs and outputs. These alterations seem to help communicate the model's intention, making it vastly more understandable for designers unfamiliar with the model. In the following section I will discuss the impact of making these alterations to parametric models used in an architecture project.

6.5 Structured Programming in Practice

Dermoid

By the third Dermoid workshop (in November 2010; fig. 51), the project team had a decided that Dermoid would consist of reciprocal hexagons formed from cambered wooden beams weaving under and over a guiding surface. The rationale for this structure is discussed in greater detail by Mark Burry (2011) in *Scripting Cultures* but for the purpose of the present discussion, suffice to say, the chosen design direction presented numerous modelling challenges. By the November workshop there were still many unknowns, including, the shape of the surface, the details of the beam joints, and the overall structural performance. These would remain unknown until days before the construction commenced in March 2011 (having been calculated progressively through a series of physical modelling experiments). The unknowns suited the flexibility of parametric modelling, yet the reciprocal frame did not lend itself to parametric modelling since distributing a pattern on a doubly curved surface is a difficult problem made harder in this instance by the circular relationships of the reciprocal frame (which lend themselves to iterative solving rather than the linear progression of a parametric model). Thus, while months of work had occurred prior to the November workshop, most parts of the parametric model were still up for negotiation and required a degree of flexibility. I took the lead in developing the models for this stage of the

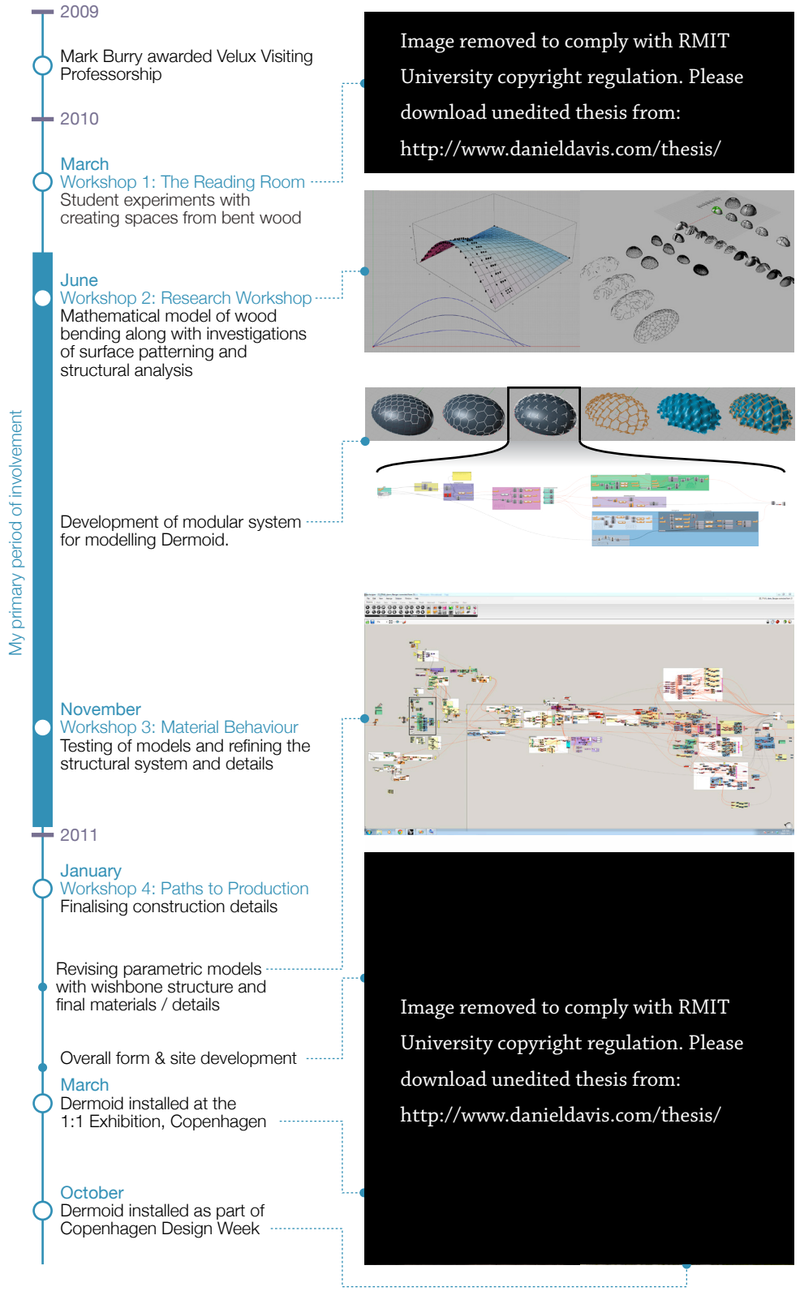


Figure 51: Key milestones in the development of Dermoid. Unlike a traditional design process, Dermoid's design commences with investigations into the material properties of wood, and proceeds through detailing and design development, before concluding with a sketch of the form. This process is enabled to a large degree by the flexibility of the parametric models.

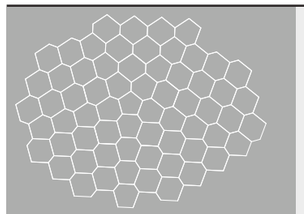
project. In order to achieve the needed flexibility, I experimented with structuring the models. Doing so allowed me to consider the practicalities of structuring parametric models during a design project, and it also allowed me to observe how the structured models evolved once I handed them to other team members.

Structuring the Project

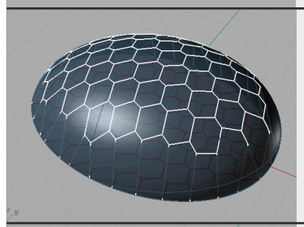
In the months prior to the November workshop, a number of key modelling tasks emerged as areas of research:

1. Distributing the pattern evenly over the doubly curved surface.
2. Calculating the intersection points of the reciprocal frame.
3. Shaping and detailing the beams.

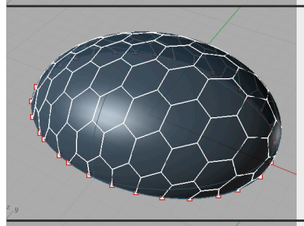
In a conventional linear design process, these considerations would come as part of *Design Development* or *Detailed Design*. It is of significance that they should be the early stages of Dermoid's design process (fig. 51). The dissociation with the orthodox design progression carries through to other stages of Dermoid's design where, for example, the construction documentation was produced prior to finalising the overall form. While changing a project's form after generating the construction documentation would ordinarily be extremely disruptive and time consuming, the flexibility of Dermoid's parametric models accommodated these types of late changes relatively effortlessly. In many ways this is the antithesis of Paulson and MacLeamy's front-loading (see chap. 2.2): rather than forcing designers to make critical decisions early in a project as a means to avoid expensive design changes, in Dermoid the cost of change is lowered to the point where critical decisions can be delayed until the designers best understand the consequences of these decisions – even if this means delaying a decision until almost the end of a project. The flexibility of Dermoid's parametric models essentially compressed the design cycle, allowing conceptual decisions to manifest quickly in construction documentation, allowing critical decisions to be delayed, and allowing the design process to begin with considerations not conventionally explored until later in the project.



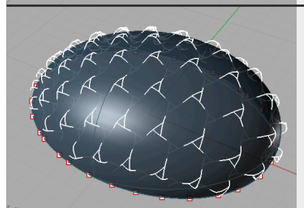
Stage-A
Function: Generate the pattern
Inputs: n/a
Outputs: 2d network of lines



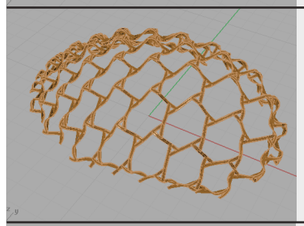
Stage-B
Function: Projects lines onto surface
Inputs: A surface; 2d lines
Outputs: A surface; 3d line pattern



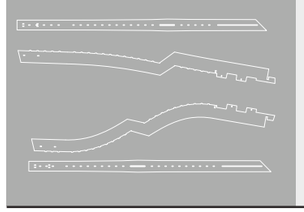
Stage-C
Function: Relaxes pattern to distribute lines more evenly
Inputs: A surface; 3d line pattern
Outputs: A surface; 3d line pattern



Stage-D
Function: Rotates each line to create the reciprocal frame and weaves the line under and over the surface to camber the beam
Inputs: A surface; 3d line pattern
Outputs: Network of curves



Stage-E1
Function: Creates flanges and webs along curves to visualise structure
Inputs: Network of curves
Outputs: Array of surfaces



Stage-E2
Function: Prepares construction documentation
Inputs: Network of curves
Outputs: Laser cutting files

Figure 52: The outputs from the chain of parametric models that generate Dermoid.

The project team developed a number of parametric models prior to the November workshop as they explored the three initial areas of research – pattern distribution, intersection points, and beam details. These models naturally form a chain (fig. 52) that progressively generates Dermoid, beginning with a two-dimensional pattern (Stage-A) and ending with the construction documentation (Stage-E2). Each stage in this chain can be thought of as a module since each has a prescribed set of inputs (from the previous stage) and a distinct set of outputs (for the next stage). The demarcations of these modules was not something imposed at the start of the project, rather they naturally emerged and crystallised around the volatile points of the project (pattern distribution, Stage-A, B & C; intersection points, Stage-D; beam details, Stage-E2). In hindsight the structure follows David Parnas's (1972, 1058) advice to decompose projects around "difficult design decisions or design decisions which are likely to change." By decomposing Dermoid around key points of research, each research question had a respective parametric model that could change to accommodate research developments. Provided any new parametric model outputted all of the stage's requisite data, changing the parametric model would not disrupt the overall project. This allowed the team members to work concurrently on different aspects of the project without interfering with each other's work. The structure was also software agnostic provided each model returned the right outputs. This proved useful on *wicked* stages (Rittel and Webber 1973) like pattern distribution (Stages B & C) where the stage's parametric model was rebuilt in at least five different software packages during the course of the design. Being able to modify stages of a project without disrupting the overall project is described by Meyer (1997, 40-46) as *continuity*. Although breaking a parametric model into six stages and manually feeding data between them may seem intuitively less flexible than using a single parametric model, the continuity offered by decomposing Dermoid into six distinct stages helped improve the project flexibility by facilitating team-work and by helping make changes less disruptive.

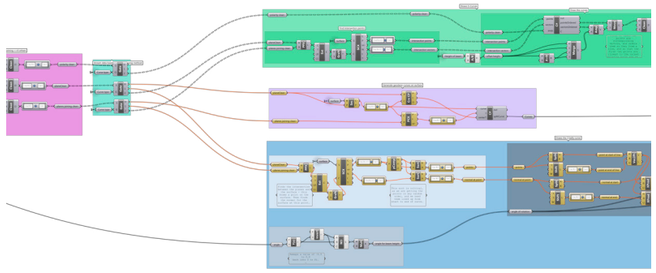


Figure 53: A section of the parametric model from Stage-D, which demonstrates the structure of the models used in the November Dermoid workshop.

Structuring the Models

I began experimenting with structuring the models driving Dermoid as I prepared them for the November workshop (fig. 53). The models had initially been created in an unstructured way. To add structure I normally had to do the following: prune the branches of code not contributing to the model's outcome; add new nodes to name paths of data clearly; and group the nodes into modules by looking for places where the data was naturally channelled into one or two streams. Software engineers call this process of restructuring code, *refactoring*. By beginning with unstructured code that I later refactored into a structured model I perhaps missed out on using structure as a compositional and decompositional design aid (Meyer 1997, 40-46), or as Green and Petre (1996, 7) put it, the "jumps from a high level to a low level or vice versa". While I normally follow the practices described by Meyer, Green, and Petre when writing textual code, I found it difficult to use structure as a guide to create these visual programs. I have experimented with teaching architecture students to create visual programs guided by the structure of Input-Process-Output diagrams (Davis, Salim, and J. Burry 2011). While this method has had modest success, particularly at getting students unfamiliar with programming to think algorithmically, structured programming still feels forced in the visual programming environment of Grasshopper. This preference for structure through refactoring unstructured models may be tied to how structure is implemented in Grasshopper, as I discussed earlier.

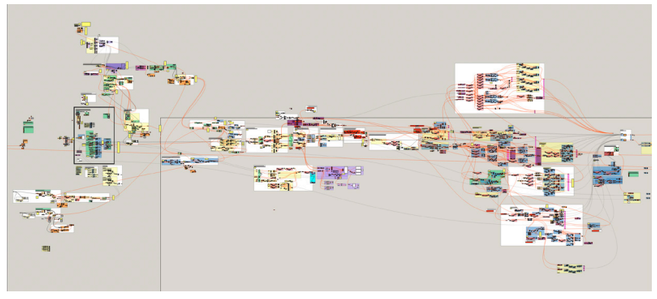


Figure 54: The final parametric model used to design Dermoid. While this model looks messy, the model's creator has actually composed the model out of a hierarchy of modules that make it relatively easy to understand the model given its size and complexity. Many of these modules are reused from earlier iterations of the project.

After Structure

After the November workshop the Danish team members took charge of finalising the parametric models as they prepared for Dermoid's construction in March 2011 (fig. 51). This allowed me to observe how the structured models fared as major changes were made to them by designers unfamiliar with the model's structure. Three critical modifications were made during this period:

1. The models in stages B and C (fig. 52) were replaced by a model in Maya, which used Maya's Nucleus engine to derive Dermoid's overall form and pattern.
2. The cambered beams were bifurcated into a wishbone structure.
3. The beam details and construction documentation were refined for the specific construction materials and methods.

The first modification (using Maya to derive the form and pattern) was simply a case of swapping models. Since the Maya model returned all the expected outputs, the continuity of the project was preserved and none of the surrounding models had to change. The other two modifications (changing the topology of the beam and altering the construction documentation) required extensive adaptations to the existing parametric models. These changes were primarily carried out by a team member who joined the project during the November workshop. While they were initially unfamiliar with the models and my rationale for structuring the models, they required very little guidance in modifying them (they seldom contacted me for assistance). In order to make the changes, the designer chose to combine all the stages of the project together into one massive model (fig. 54). The resulting model contains 4086 nodes, which makes

it twice as large as the largest model from chapter 4.3 and approximately two hundred times larger than the average Grasshopper model. Without prompting from me, the designer had carefully composed the model from a hierarchy of modules. Almost all of the modules from the original model had been reused, and these were complemented with a large number of new modules the designer had created. The reuse of the modules demonstrates that the designer could understand them well enough to apply them in a new context, despite being initially unfamiliar with the project. While the modules were cumbersome to create, this type of reuse demonstrates clear benefits to structuring a project in terms of improving understandability, collaboration, and reuse.

The complexities of Dermoid, both in terms of geometry and in terms of collaboration, place it on the limit of what is currently possible in parametric modelling – and perhaps beyond what is practical with an unstructured visual program. Breaking the project into a hierarchy of stages seemed to make it possible for designers to collaborate using disparate software, while the modules within the models seemed to promote model reuse and improve model understandability. At both scales, structure was difficult to impose at the start of the project and instead tended to emerge from an unstructured beginning to be later refactored with a few relatively minor changes. Perhaps most significantly, the flexibility of this working method facilitated the reorganisation of the design process, which enabled the designers to delay critical decisions until they had the best understanding of their consequences, rather than forcing the decisions early in order to avoid the cost of later changes.

6.6 Sharing Modules Online

By structuring Dermoid's parametric models I had amassed a library of modules able to be reused on other projects (as they were in later versions of Dermoid). In order to share these modules with other designers, I created the website parametricmodel.com, which lets anyone download and use the modules under the Creative Commons Attribution-ShareAlike licence (2007). The pages for each module intentionally resemble the documentation programmers provide with libraries of modules; the page for each module starts out with a short blurb, notes the modules inputs and outputs, and then enters into a detailed description of how the module

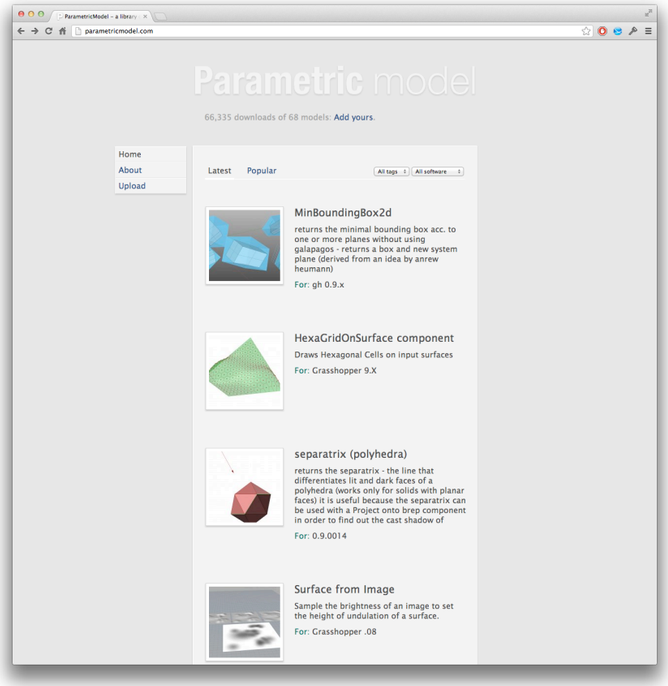


Figure 55: The homepage of parametricmodel.com as of 5 January 2013.

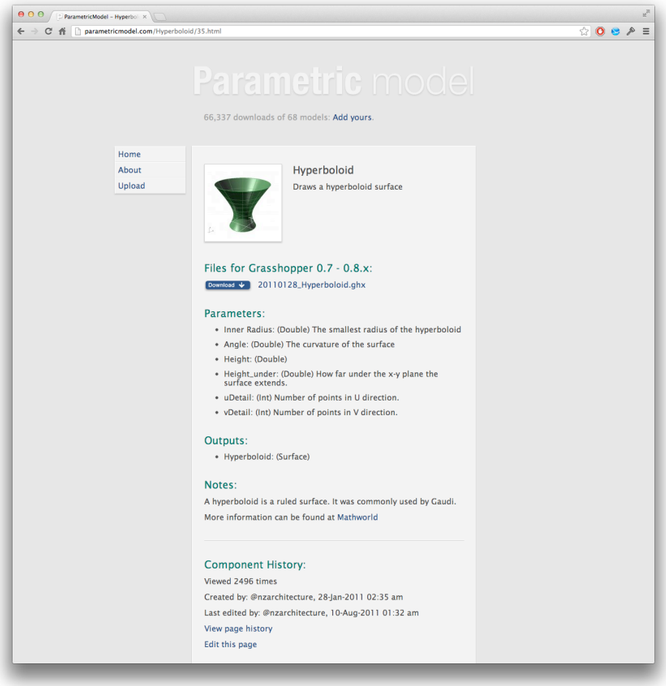


Figure 56: T he hyperboloid module download page. Like documentation that comes with many programming languages, the download page details what the module does, the parameters it requires, and the outputs it produces.

works. At the time of writing, July 2012, parametricmodel.com has been running for twenty months, and in that time the 57 modules on the site have been downloaded 47,740 times by 19,387 people from 127 countries.[13] While I do not have access to the projects the modules have been reused on, the 47,740 downloads indicate that the modules are reusable in a wide range of contexts and useful to a large number of designers.

With the success of the module downloads, I was also interested in whether parametricmodel.com could encourage designers to modularise their models and share them via the website. When I launched parametricmodel.com I designed the site so anyone could upload and share a module. In the module upload page I attempted to balance prescriptively enforcing a modular structure while minimising the obstacles to uploading. As such, the website coaxes users into creating modules by asking them to describe uploaded models with modular programming principles: defining the inputs and the outputs, describing the problem the module solves, and explaining how the module works. This has been relatively successful with all the uploaded models conforming to the modular pattern. Yet for all the modules downloaded, very few have been uploaded; for every thousand people who download a module, on average only one returns to contribute a new module. There are a whole host of reasons why architects may be reluctant to upload modules, which range from concerns about liability, to the effort and skill required in packaging a module, to a preference for contributing to other websites – particularly personal websites – where they may receive more control and more recognition. Despite the failure of parametricmodel.com to elicit a large number of contributions, it has been successful in demonstrating that thousands of designers want to reuse pre-packaged modules. As was shown with the Dermoid project, structure contributes to the reusability of components both by making them more understandable and by making them easier to extract for sharing. While structure may encourage sharing, there are other factors involved, including, intellectual property rights and the intrinsic rewards individuals receive for sharing. Parametricmodel.com shows how designers may benefit if these impediments are overcome, and the creation and sharing of modules becomes more widespread.

13 The ten most active countries being: United States, United Kingdom, Germany, Australia, Italy, Austria, Spain, France, Russia, and the Netherlands.

6.7 Conclusion

Out at the edge of Copenhagen, out past the Royal Danish Academy of Fine Art, and in the secluded concrete studio filled with researchers from CITA and SIAL, we were facing a deceptively simple problem. The problem was not directly the one posed earlier – how can we fashion a doubly curved pavilion from a wooden reciprocal frame? With time the numerous difficulties of this proposition were solvable. Rather the deceptively simple problem was keeping the project flexible long enough for these discoveries to be made. Structuring Dermoid's parametric models undoubtedly improved the project's flexibility, enabling knowledge of Dermoid's form and material strength to inform the project just days before construction.

I say the problem is deceptive because a model's structure is not necessarily an obvious contributing factor to a project's flexibility. Indeed, during the 1960s' software crisis many software engineers overlooked the importance of program structure, often instinctively believing their woes were a product of perceived inadequacies in areas like project management. Today, however, structure is seen as so pivotally important to successful programming that even basic introductions to programming normally involve learning about structure, and some modern programming languages mandate the use of structure. Yet architects creating parametric models with visual programming languages are given only rudimentary tools for structuring projects and receive almost no guidance in the educational material on how to structure a project (one exception being Woodbury, Aish, and Kilian [2007] giving the structural recipes for common architecture problems). It is therefore not surprising that the majority of architects overlook something as deceptively simple as clearly naming parameters (81% do not name parameters) or using clusters in their Grasshopper models (97.5% do not use Grasshopper's inbuilt modular structure, clusters; see chap. 6.3).

The widespread omission of structure in models created by architects makes for concerning statistics in light of the benefits structure provides. My thinking-aloud interviews seem to suggest that structure largely determines whether an architect can understand a model, which is a finding that confirms the existing research on the cognition of professional programmers. Yet using structure to cognitively jump "from a high level to a low level or vice versa" (Green and Petre 1996, 7) – such as professional

programmers often do – proved to be difficult in the visual programming environments used by architects. From my experience structuring Dermoid's parametric models, structure came from refactoring unstructured models rather than being the scaffold onto which programs are decomposed and composed as Meyer (1997, 40-46) suggests. Nevertheless, breaking Dermoid into a hierarchy of modules made it possible for designers to collaborate using disparate software, and offered them the continuity to make radical changes late in the project. The degree of flexibility within this structure challenged the orthodox progression of the design process, enabling details to be examined much earlier whilst allowing ordinarily pivotal decisions to be explored right up until the point of construction. In essence this was the antithesis of Paulson and MacLeamy's front-loading (see chap. 2.2): rather than making decisions early in order to avoid the expense of changing them later, in Dermoid the cost of change was lowered to the point where critical decisions could be delayed until they were best understood. The structure also enabled parts of the models to be extracted and reused by designers initially unfamiliar with the models. While structure potentially encourages reuse, parametricmodel.com shows sharing requires more than an easily decomposed structure. These benefits of structure – in terms of reuse, understandability, continuity, and design process flexibility – remain largely unrealised by architects. While this is concerning, structure can be introduced with a few simple alterations. The most effective strategies seem to be clearly naming parameters, and grouping nodes together by function with defined inputs and outputs. I have posited in this chapter that architects do not realise the benefits of these simple structural changes due to both the limitations of design environments and the way architects are educated, an argument I will pick up again in the discussion.

7 Case C: Interactive Programming

First iteration: Responsive Acoustic Surfaces.
Location: SmartGeometry 2011, Copenhagen, Denmark.
Project participants: Mark Burry, Jane Burry, John Klein, Alexander Peña de Leon, Daniel Davis, Brady Peters, Phil Ayres, Tobias Olesen.

Second iteration: The FabPod.
Location: RMIT University, Melbourne, Australia.
Project participants: Nick Williams, John Cherrey, Jane Burry, Brady Peters, Daniel Davis, Alexander Peña de Leon, Mark Burry, Nathan Crowe, Dharman Gersch, Arif Mohktar, Costas Georges, Andim Taip, Marina Savochina.

Code available at: yeti3d.com (GNU General Public Licence)

Related publications:

Davis, Daniel, Jane Burry, and Mark Burry. 2012. "Yeti: Designing Geometric Tools with Interactive Programming." In *Meaning, Matter, Making: Proceedings of the 7th International Workshop on the Design and Semantics of Form and Movement*, edited by Lin-Lin Chen, Tom Djajadiningrat, Loe Feijs, Simon Fraser, Steven Kyffin, and Dagmar Steffen, 196–202. Wellington, New Zealand: Victoria University of Wellington.

Burry, Jane, Daniel Davis, Brady Peters, Phil Ayres, John Klein, Alexander Peña de Leon, and Mark Burry. 2011. "Modelling Hyperboloid Sound Scattering: The challenge of simulating, fabricating and measuring." In *Computational Design Modeling: Proceedings of the Design Modeling Symposium Berlin 2011*, edited by Christoph Gengnagel, Axel Kilian, Norbert Palz, and Fabian Scheurer, 89-96. Berlin: Springer-Verlag.

Figure 57: John Klein (left) and Alexander Peña de Leon (right) in a Copenhagen hotel at 3 a.m. writing code six hours before the start of SmartGeometry 2011.

7.1 Introduction

To be writing code in Copenhagen at 3 a.m. was not an unusual occurrence. We had spent the past week in Copenhagen sleeping only a couple of hours each night as we rushed to get ready for SmartGeometry 2011. It turns out casting plaster hyperboloids is hard, much harder than the model makers from Sagrada Família make it look. And it turns out joining hyperboloids is hard, much harder than Gaudí makes it seem.[1] When figure 57 was taken, we were just six hours away from the start of SmartGeometry 2011, and we were all exhausted from days spent fighting with the geometry and each other. So naturally, rather than verify the plaster hyperboloids joined as expected, we went to sleep for a couple of hours.

Sleeping is a decision we would come to regret two days later. The workshop was half way through and we had cut the formwork for roughly forty hexagonal plaster hyperboloid bricks, when we realised none of the hyperboloids joined together. Instead of sitting flush against one another, the brick's wooden sides were angled such that they could only join together if there were slight gaps between the bricks. The error was small (less than 5mm on a brick 450mm wide) but these small errors accumulated through the stacking of the bricks, which caused visible gaps in the upper courses and prevented the topmost courses coming together at all (fig. 58). Without the time to recut the formwork, that single small

1 The Responsive Acoustic Surface was built to test the hypothesis that hyperboloid geometry contributed to the diffuse sound of the interior of Antoni Gaudí's Sagrada Família. For more information about the rationale for using hyperboloids in the Responsive Acoustic Surface and for more information about the surface's acoustic properties, see *Modelling Hyperboloid Sound Scattering* by Jane Burry et al. (2011).

Figure 58: The responsive acoustic surface installed at SmartGeometry 2011. Slight gaps are still visible in the top- and bottom-most rows.

Figure 59: The responsive acoustic surface (foreground) with its counterpart, the flat benchmark surface in the background. The top of the responsive acoustic surface curves slightly backwards due to a small error in the shape of the brick. Bolts between the plywood frames help to pull the bricks together but also put the frame under a lot of stress.

error threatened the whole viability of the project. Once again we were without sleep. Thankfully the timber formwork had enough pliability to accommodate the error, although if you look closely at figure 59 you can see the timber is under such tension the whole wall bows slightly backwards.

This single small error can be attributed to exhaustion: in our fatigued state we did not verify that the code's math generated the expected intersections between hyperboloids. This could be seen as a failing of our project management skills – a failure to allocate sufficient time to verify the code's outputs. But it could also be seen as a failing of the coding environment – a failure to provide immediate feedback about what the math in the code was producing. The notion that programming environments fail to provide designers with immediate feedback forms the foundation of Bret Victor's (2012) manifesto, *Inventing on Principle*. Victor, a user experience designer best known for creating the initial interface of Apple's iPad, describes how the interface to most programming environments leaves the designer estranged from what they are creating:

> Here's how coding works: you type a bunch of code into a text editor, kind of imagining in your head what each line of code is going to do. And then you compile and run, and something comes out… But if there's anything wrong, or if I have further ideas, I have to go back to the code. I go edit the code, compile and run, see what it looks like. Anything wrong, go back to the code. Most of my time is spent working in the code, working in a text editor blindly, without an immediate connection to what I'm actually trying to make.
> *Victor 2012, 2:30*

In this case study I follow a similar line of thinking, observing that typically for architects there is a significant delay between editing code and then, much later, realising your plaster hyperboloids do not fit together as expected. As such, I use this case study to consider how coding environments could provide architects with more immediate feedback about what their code produces. I begin by discussing the history of interactive programming and the lack of interactive programming environments for architects. I then describe an interactive programming environment I created, dubbed *Yeti*, and compare Yeti's performance to two existing coding methods on three architecture projects (including revisiting the plaster hyperboloids of the Responsive Acoustic Surface). But first I want to return to Bret Victor's manifesto.

7.2 The Normative Programming Process

In *Inventing on Principle*, Bret Victor (2012, 2:30) describes the normative programming process as, "edit the code, compile and run, see what it looks like." This sequence of events is commonly known as the *Edit-Compile-Run* loop. In the loop (fig. 60), the programmer *edits* the text of the code [1], presses a button to activate the code [2], and then waits. They wait first for the computer to validate the code [3], then they wait for the computer to *compile* the code into machine-readable instructions [4], and finally they wait for the computer to *run* this set of instructions [5]. Only then can the programmer see what their code produces [6]. Victor (2012, 18:00) says that good programmers shortcut this process by mentally simulating the running of code – a somewhat perverse situation considering they are more often than not sitting in front of a machine dedicated to doing just that.

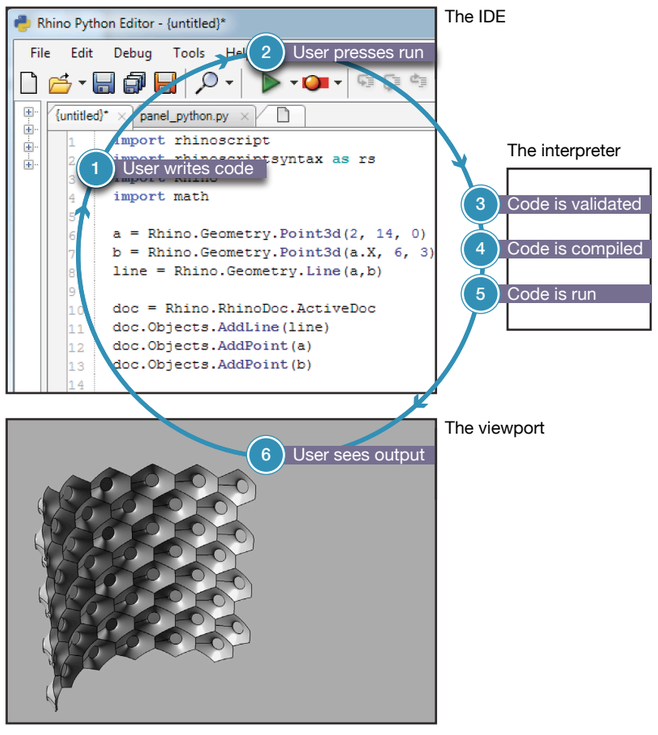


Figure 60: The Edit-Compile-Run loop for a Rhino Python script. A designer must go through this loop every time they want to see what their code produces. In the best case it takes a couple of seconds to move between writing code [1] and seeing the output [6] but this period can be much longer if the script is computationally intensive to run.

For architects, the delayed feedback from the Edit-Compile-Run loop proves problematic. Ivan Sutherland (1963, 8) disparagingly called this "writing letters to rather than conferring with our computers." Yet shortcutting this process using mental simulation, as good programmers often do, clashes with Nigel Cross's (2011, 11) observation that "designing, it seems, is difficult to conduct by purely internal mental processes." Cross's contention that designers need to have continual feedback, separate from their internal monologue, is shared by Sutherland (1965), Victor (2012), and many others (Schön 1983; Lawson 2005; Brown 2009). This view is reinforced by design cognition research that shows any latency between a designer's action and the computer's reaction is problematic for architects since delays exacerbate change blindness, which makes it hard for designers to evaluate model changes (Erhan et al. 2009; Nasirova et al. 2011; see chap. 2.3). With designers potentially blind to the changes they make, Rick Smith (2007, 2) warns that a change to a parametric model "may not be detected until much later in the design phase, or even worse, in the more expensive construction phase." Smith's warning rings true with the hyperboloid bricks of the Responsive Acoustic Surface, where feedback from a coding error was not apparent until the bricks were stacked during the construction phase.

The Edit-Compile-Run loop prevails, argues Victor (2012, 28:00), because most programming languages "were designed for punchcards" where "you'd type your program on a stack of cards, and hand them to the computer operator, and you would come back later" – an "assumption that is baked into our notions of what programming is." While punchcards may explain the origins of the Edit-Compile-Run loop in programming, there have been many developments in programming since the days of punchcards. In particular, significant developments have been made to the tools programmers use to write code, known as Integrated Development Environments (IDEs). Modern IDEs often augment the Edit-Compile-Run loop so programmers do not have to wait for feedback. For example, some IDEs identify simple logical errors before the code is run, and some IDEs suggest and explain programming commands while programmers are writing them (a feature known as autocompletion). Other IDEs allow the basic editing of running code, which enables programmers to make minor changes without cycling back through the edit-compile-run loop (this is known as interactive debugging). These types of IDE features makeup part of Section 2.3.1 of the *Software Engineering Body of Knowledge Version 1.0*

(Hilburn et al. 1999); a section of knowledge that often reinforces the notion that programming languages were designed for punchcards, while also offering ways of making Edit-Compile-Run loop more interactive.

The interactive feedback mechanisms of many modern IDEs have not filtered down to the environments architects write code in. Like professional programmers, architects use languages based on the Edit-Compile-Run loop, with Leitão, Santos, and Lopes (2012, 146) pointing out that even popular languages like "RhinoScript are a descendant of a long line of BASIC dialects that started much earlier, in 1964." But unlike professional programmers, who have the advantages of cutting edge IDEs, Leitão, Santos, and Lopes (2012, 143) say that in the context of architecture "the absence of a (good) IDE requires users to either remember the functionality or read extensive documentation." Thus architects are left to contend with the historic Edit-Compile-Run loop without many of the interactive conveniences present in the IDEs used by modern software engineers. This lack of interactivity in the programming process causes pronounced latency between the designer writing code and the computer generating the geometric results, which makes evaluating code changes potentially difficult for architects.

7.3 The Interactive Programming Process

Interactive programming (also known as live programming) seeks to remove any latency between writing and running code. Instead of a programmer activating the Edit-Compile-Run loop every time they want to see what their code produces, a programmer using interactive programming directly changes the code of an already running program.[2] Bret Victor (2012) demonstrates interactive programming with a programming environment he created for drawing and animating two-dimensional objects (fig. 61). When Victor changes code in the text editor, the corresponding image produced by the code changes instantly – without Victor manually pressing

2 Interactive programming is primarily about changing code while it runs. Although this is useful for displaying code changes in real time, there are many other uses for interactive programming. A common use case is to update software that cannot be shut down (for example, life support systems and certain infrastructure systems). Instead of compiling and running a new instance of the software, software engineers can use interactive programming to apply code changes to the existing software while it runs.

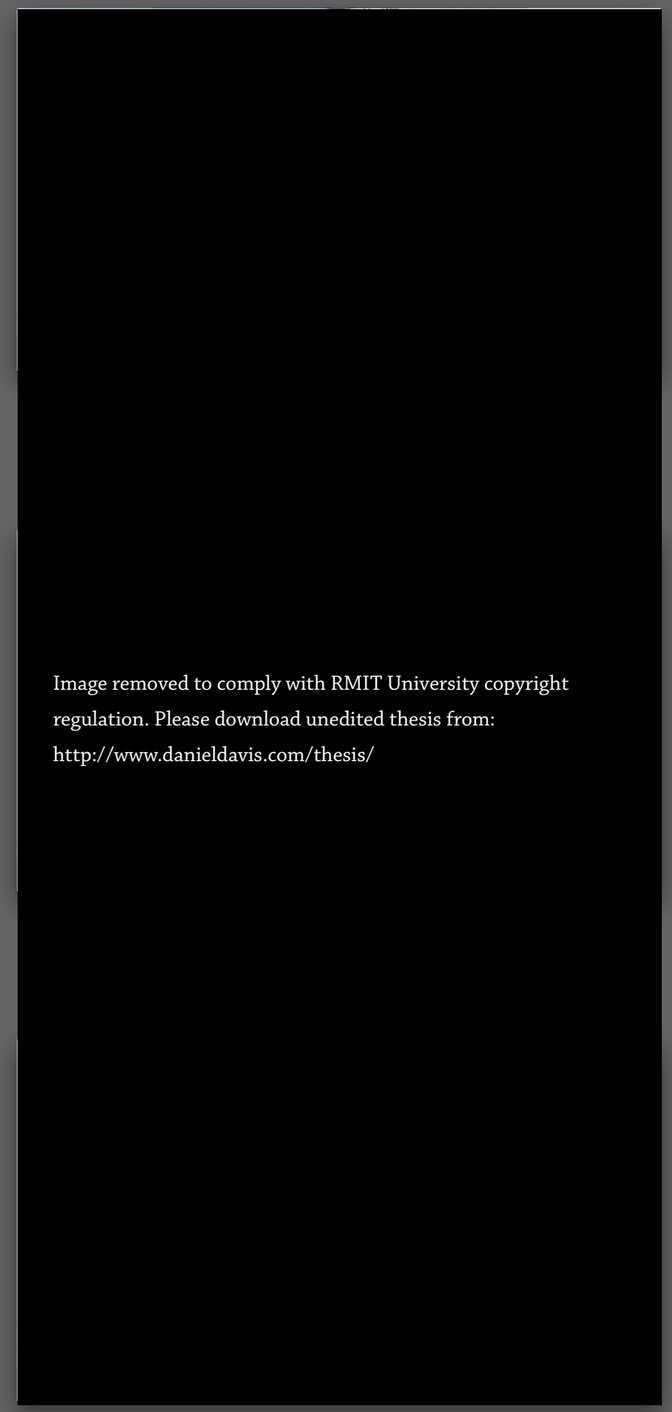


Figure 61: Bret Victor's (2012) IDE from *Inventing on Principle*. Since the programming environment is interactive, the code and the image are always in sync. As shown in the three frames, changes to the code also immediately change the image produced by the code – without the designer manually activating the Edit-Compile-Run loop to see them.

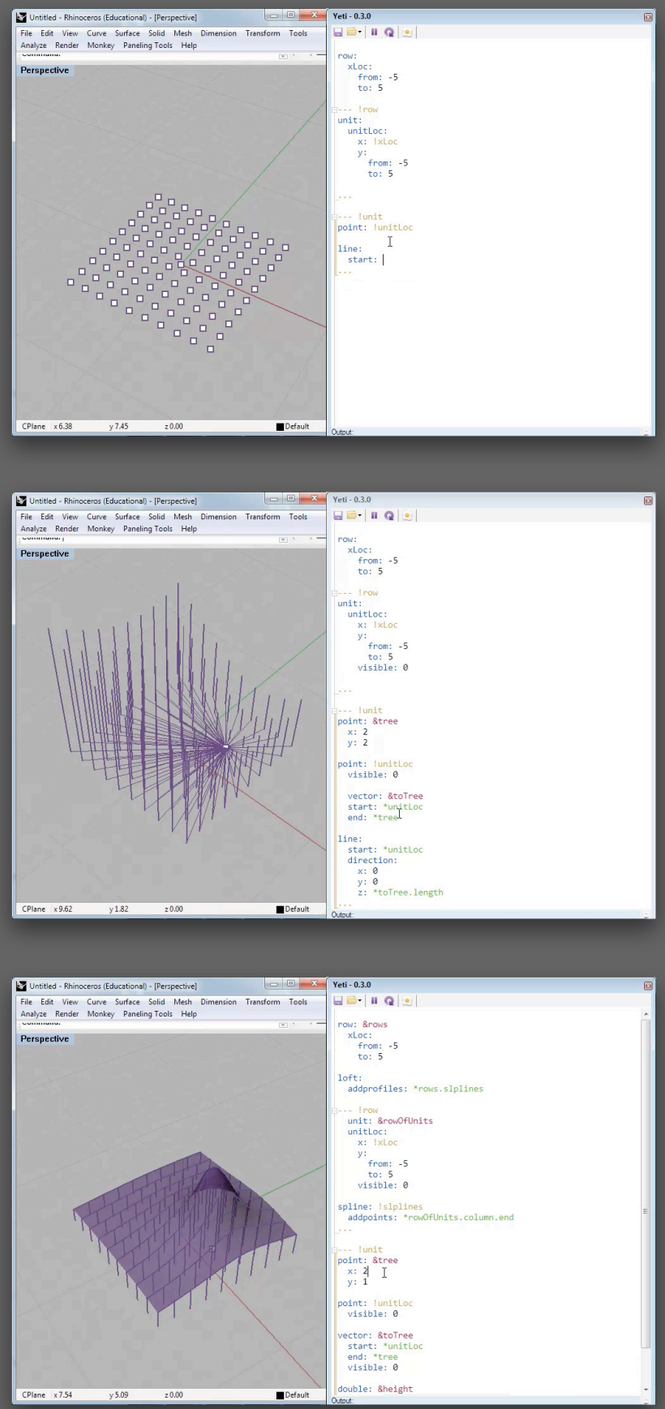


Figure 62: Yeti, an interactive programming plugin for Rhino. Like with Victor's IDE (fig. 61), the code and the model are always in sync. Whenever the code changes, the model produced by the code automatically changes as well.

a button to execute the Edit-Compile-Run loop. With the code always in sync with the image it produces, Victor (2012, 2:00) argues that his environment gives designers "an immediate connection to what they are creating."

To assist architects creating parametric models I developed an interactive programming environment named Yeti (fig. 62). On first glance Yeti looks similar to Victor's interactive programming environment, however, there are a number of key differences between the two IDEs. The most obvious difference is that Victor's environment focuses on the real-time drawing of two-dimensional objects, while Yeti supports the real-time remodelling of three-dimensional objects (since three-dimensional objects generally require more computational resources, making the calculations in real time is significantly more challenging than with two-dimensional objects). The second significant difference is that Victor's presentation of his environment in January 2012 comes a number of months after I first presented Yeti, and released Yeti's source code, in May 2011. While Victor was not the first to create an interactive programming environment, I have chosen to cite him both because he clearly articulates the problems with the normative programming process, and because his legacy of creating interfaces for Apple adds credibility to the argument that interactive environments, like Yeti, are important emerging areas of research for designers.

While Yeti predates Victor's programming environment by a couple of months, a number of other interactive programming environments predate both of them by many years. The origins of interactive programming date back to the programming languages LISP (first version: 1958) and SmallTalk (first version: 1971), both of which allow programmers to modify code while it runs. The initial emphasis was on updating software without needing to shut down running programs (useful for critical systems). These techniques were extended, in particular by musicians, to allow the real-time modification of code. For musicians, these interactive programming environments enable them to modify code driving musical compositions whilst immediately experiencing the modification's sonic implications. The first performance with an interactive environment was by Ron Kuivila at STEIM in 1985 using the FORTH programming language (Sorensen 2005). In the early 1990s, interactive textual programming environments gave way to interactive visual programming environments like Max/MSP (the

precursors of the visual programming environments architects use today). While visual programming remains popular with musicians, a number of new interactive textual programming languages have emerged, including the Smalltalk based Supercollider (McCartney 2002) as well as the LISP based ClunK (Wang and Cook 2004) and Impromptu (Sorensen 2005). Outside the domain of music there is a scattering of interactive programming environments aimed at designers, such as SimpleLiveCoding for Processing and the widely used Firebug for CSS editing. While real-time interactive programming suits these creative contexts, the computational stress of three-dimensional design has meant that architects – prior to my research – have been unable to utilise interactive programming.

The crux of all interactive programming environments is removing the latency between writing and running code. Existing interactive programming environments achieve this in a number of ways:

- **Automation** Rather than waiting for the user to manually tell the Edit-Compile-Run loop to execute, the loop can be set to run automatically and display the results whenever the code is changed – as is done in SimpleLiveCoding (Jenett 2012). This is a bit like stop-motion animation; the user sees a single program adapting to code changes but really they are seeing a series of discrete programs one after the other (like frames in a movie). In order for this animation to feel responsive, the elapsed time between the user changing code and the completion of the Edit-Compile-Run loop should ideally be a tenth of a second and certainly not much more than one second (Miller 1968, 271; Card, Robertson, and Mackinlay 1991, 185). For simple calculations these time restrictions are manageable. However, for complicated calculations it becomes impractical to recompile and recalculate the entire project every time the code changes, especially if the change only impacts a small and discrete part of the finished product.

- **Sequencing** For musicians using interactive programming, changes must happen relative to an underlying time signature. Code from Supercollider (McCartney 2002), ClunK (Wang and Cook 2004), and Impromptu (Sorensen 2005) all generate timed sequences of actions for the computer to perform. As the code changes, new actions are automatically queued into the sequence while old actions are seamlessly discontinued (Sorensen 2005), which avoids the stopping and restarting necessary when using the Edit-Compile-Run loop. This method has been adapted to generate simple geometry in time to music (Sorensen and Gardner 2010, 832). However, for architects doing computationally demanding geometric calculations, generating geometry rhythmically is not as important as generating geometry quickly. For this reason, sequencing is unsuitable in an architectural context.

- **Hot-Swapping** The Edit-Compile-Run loop recompiles every line of code even if some lines have not changed since the last time the loop was activated. Instead of compiling every line of code, hot-swapping allows small chunks of code to be independently compiled and then integrated with the unchanged parts of the program – while the overall program continues to run. This reduces the latency of compilation but does not reduce the latency of running the code.[3] Since geometric calculations take orders of magnitude longer than the compilation of code, the savings from hot-swapping in an architectural context are likely comparable to those of *automation*.

Although there are a range methods for reducing the latency between writing and running code, none of the existing methods are suited to the unique challenges of performing geometric calculations in real time. These are challenges not present in other design disciplines currently using interactive programming (such as web-design, musical performance, and two-dimensional animation). Despite the range of textual interactive programming environments available to other designers, architects currently have no option but to contend with the separation induced by the Edit-Compile-Run loop.

3 Although the code continues to run when it is hot-swapped, there is no way of easily knowing how the hot-swapped code transforms the geometric model. Therefore, to update the model, the code must be rerun, which means hot-swapping in this context only saves compilation time and not running time.

7.4 Interactive Visual Programming

While none of the existing interactive textual programming environments are particularly suited to architects, many non-textual programming environments allow the interactive creation of geometry. Grasshopper, Houdini, and GenerativeComponents all overcome the problem of performing geometric calculations in real time by representing geometric relationships with a Directed Acyclic Graph (DAG) (Woodbury 2010, 11-22). As explained in chapter 5.3, a DAG is a type of flow-chart where nodes represent geometric operations and directed edges represent the flow of data between pairs of nodes. When a node is changed, the model is updated by propagating the changes along the directed edges to update the affected nodes. This minimises the calculations involved in updating the model since the only nodes recalculated are those affected by the change. Rather than recalculating every geometric operation, as with the Edit-Compile-Run loop, the selective updating of a DAG saves unnecessary geometric calculations, greatly compressing the time between writing and running code.

While visual programming enables architects to work interactively, there are still limitations when compared to textual programming. In the previous chapter (chap. 6) I demonstrated that visual programming environments do not support structure as elegantly as many textual programming environments do. Partly citing my research from the previous chapter, Leitão, Santos, and Lopes (2012, 160) conclude, "learning a textual programming language takes more time and effort than learning a visual programming language, but this effort is quickly recovered when the complexity of the problems becomes sufficiently large." I suspect visual programming is easier to learn partly because the interactivity of visual programming provides the continuous feedback Green and Petre (1996, 8) say novice programmers require. While interactive visual programming languages are successful in the domain of architecture, there remains an opportunity to create an interactive textual programming language that combines the structural benefits of textual programming with the ease of use brought about by the interactivity of visual programming.

7.5 Introducing Yeti

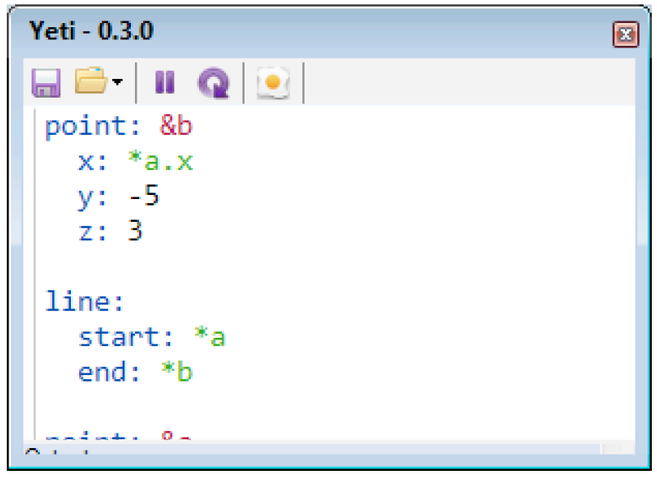


Figure 63: The Yeti interface. The primary element is a textbox for writing code. The code within the textbox is automatically coloured: numbers (black), geometry (blue), names (red), references to named geometry (green). Above the textbox are a row of icons, from left to right: save, open, pause interactive updating, force update, and bake geometry (export to Rhino). The geometry created by the code is displayed in another window (not shown).

Yeti is an interactive textual programming environment I developed to support the creation of three-dimensional geometry. At first glance Yeti looks much like any other textual programming environment, a large textbox for writing code is positioned underneath a horizontal menu of icons (fig. 63). The icons give the only outward hint of Yeti's interactive behaviour: instead of an icon for running the code there is an icon for pausing Yeti's continuous evaluation of the code. Beyond these icons the difference between Yeti and other IDEs only really becomes apparent when the designer begins writing code. Rather than writing a block of code and then pressing the *run* icon to see geometry generated by the code (through the Edit-Compile-Run loop), designers writing code in Yeti always see what their code generates. The geometry is in sync with the code that produces it, so every time the code changes the geometry automatically changes as well.

In order to perform geometric calculations fast enough that they feel interactive, Yeti employs a DAG to coordinate recalculating the geometry. This is essentially the same concept powering the interactivity of the visual programming environments Grasshopper, Houdini, and GenerativeComponents. However, Yeti's DAG is not generated through a visual interface, rather it is defined textually through the relational data format YAML.

The YAML Language

YAML's inventor, Clark Evans (2011), describes YAML as a "human friendly data serialization standard." While YAML is technically a data format, Yeti uses it like a dataflow programming language to describe the structure of a DAG.[4] As such, Yeti's code is paradigmatically distinct from the procedural programming languages that underlie most other textual programming environments used by architects (see chap. 5.2). Yeti may therefore seem initially unfamiliar to designers versed in procedural programming. However, YAML's relatively minimal syntax is fairly easy to pickup.

YAML comprises primarily of `key:value` pairs. The key is always assigned the value following the colon. For example, the code to assign a variable the value of 10 is:

```
variable: 10
```

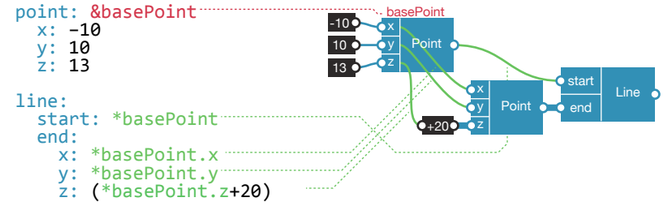
More complicated values are assigned through a list of `key:value` pairs that are separated from the parent key with indentation. For example, a point at the coordinate (-10,10,13) can be written as:

```
point:
  x: -10
  y: 10
  z: 13
```

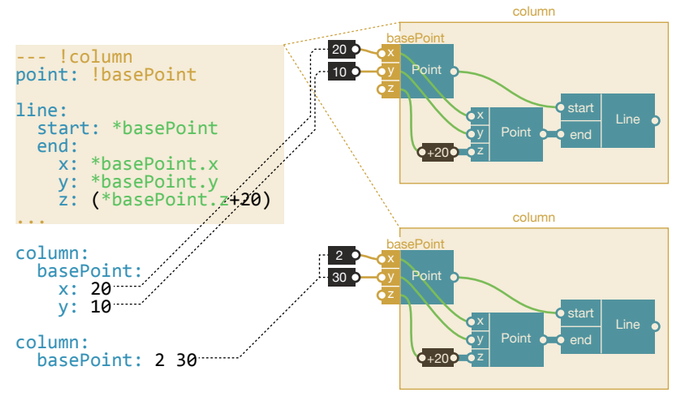
These `key:value` pairs map directly into a DAG. Every key represents a node in the graph, while values express either a property of the node, or a relationship to another node. For example:

4 By itself YAML is not a programming language since it describes data rather than computation (the concept of Turing completeness is therefore not applicable to YAML). But in Yeti this distinction is blurred because the YAML data describes the structure of a DAG, which in turn does computation. Some will say YAML is a programming language in this context, others will say it is still a data format.

Relationships between geometry can be established by labelling keys with names beginning with an ampersand [`&`] and then referencing their names later (references begin with an asterisk [`*`]). For example, the following code creates a point named `basePoint` and generates a line extending vertically from `basePoint`:



Yeti extends the YAML language so that designers can define new keys. To create a key, a designer must first create a prototype object for the key. The prototype object begins with the YAML document marker [`---`] immediately followed by the name of the object (names start with an exclamation mark [`!`]). Under this header, the designer defines the geometry of the prototype object. Any geometry given a name starting with an exclamation mark [`!`] becomes a parameter of the object that can be specified when the object is instantiated. The end of the object is delimitated by the YAML document marker [`...`]. For example, the code from the preceding example can be turned into an object named `column` and then used to generate two columns starting at (20,10,0) and (2,30,0), like so:



The user defined keys help structure the Yeti code. Like a module they encapsulate code with defined inputs and outputs (denoted by the exclamation mark [!]). However, they go further than the modules discussed in chapter 6 by providing more object-oriented features like instantiation (the creation of multiple instances that draw upon the same prototype) and inheritance (one user defined key can be based on another user defined key). In essence, YAML allows Yeti to mix the structure of textual languages with the performative benefits of directed acyclic graphs.

The Yeti Development Environment

There are a number of other interactive features in the Yeti development environment. Many of these are commonly part of the IDEs software engineers use but, according to Leitão, Santos, and Lopes (2012, 143), they are seldom a part of the IDEs architects use. The following describes some of Yeti's main interactive features.

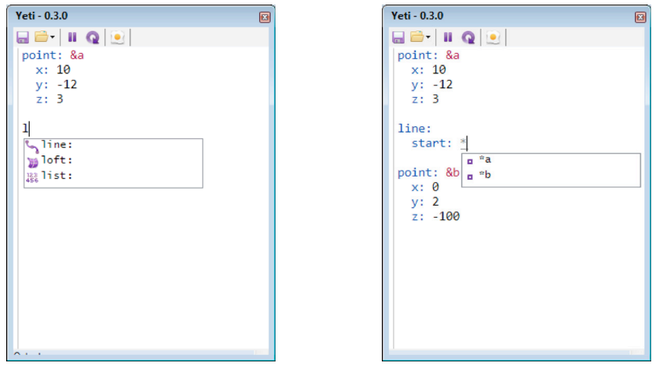


Figure 64: Yeti offering autocomplete suggestions as the designer types. Left: After the designer types the letter *L*, Yeti lists all the keys that start with the letter *L*. When the designer selects a key, Yeti will then suggest parameters for that key. Right: The designer begins typing a reference and Yeti produces a list of names used in the code.

Autocompletion:

As a designer types, Yeti predicts what the designer is typing and suggests contextually relevant keys, names, and objects. This saves the designer from looking up keys and parameters in external documentation.

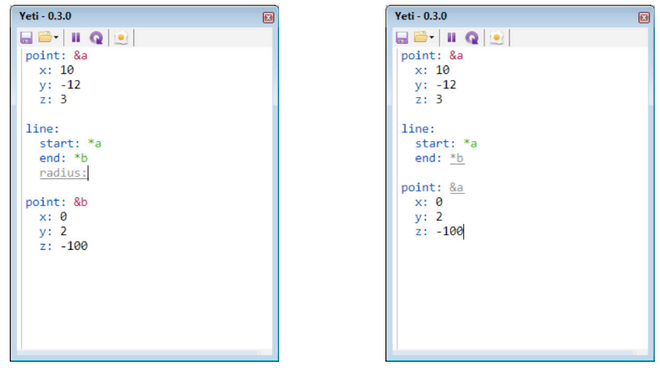


Figure 65: Errors in Yeti are coloured grey and underlined. In both of these examples, Yeti continues to function even though there are errors in the code. Left: Radius is not a valid parameter for a line, so it is marked as an error. Right: Since there is no key named &b, the pointer *b is marked as an error; and because there are two keys named &a, the second one is highlighted as an error.

Robust Error Handling:

Yeti highlights errors as they are written (common errors include spelling mistakes, syntax errors, and duplicate names). Errors generally cause procedural languages to stop running because there is no clear way to progress past an error in a sequence of instructions. Since Yeti does not use a sequence of instructions but rather a dataflow language, Yeti can continue to run code that contains errors by only parsing the error-free portions of the code into a DAG. This is important in an interactive context since evaluating code while it is being written often requires evaluating incomplete code that contains errors.

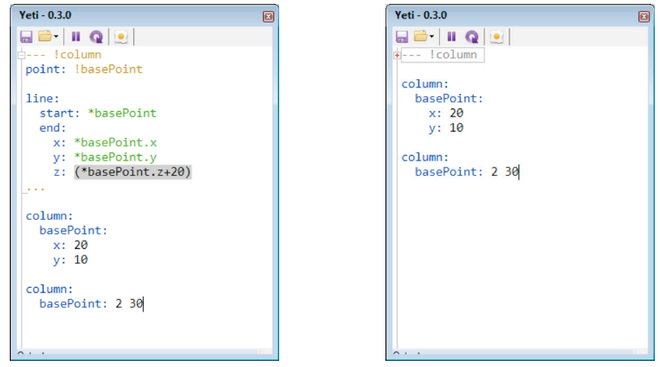


Figure 66: Left: The column object's code expanded. Right: The column object's code collapsed (the code is hidden but still functioning).

Code Folding:

The code for a prototype object can fold into a single line, effectively hiding it. These folds allow the user to improve juxtaposability by hiding irrelevant parts of the code while exposing the parts currently important.

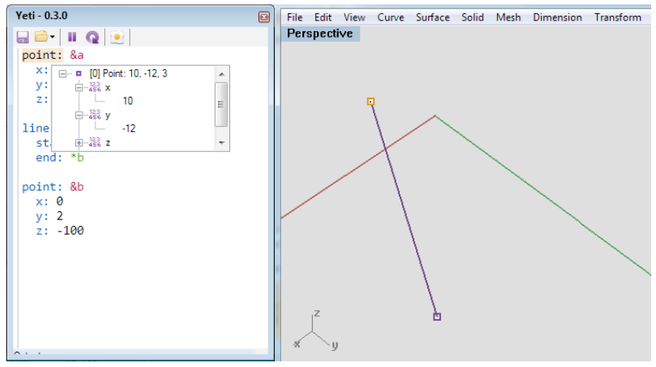


Figure 67: Clicking on the word `point:` in the code produces a window allowing the designer to inspect all the properties and parameters of the point. At the same time, the selected code and the corresponding geometry are highlighted in orange.

Interactive Debugging:

To help clarify the often-enigmatic connection between code and geometry, clicking on any key in Yeti highlights the geometry controlled by that key. A parameter window is also generated so that the user can drill down and inspect all the properties driving the geometry. Similarly, clicking on any referenced name highlights where the reference comes from in the code and the geometry it refers to. Yeti is able to provide all this information since the keys in the YAML code are directly associated with parts of the model's geometry via nodes in the DAG.

The impediments to generating geometry with interactive programming are overcome in Yeti by employing a DAG to manage geometric calculations. The DAG helps reduce the latency between writing code and seeing the geometry produced. Furthermore, the DAG also helps power other interactive features like robust error handling and interactive debugging. In the following pages I consider how these features perform when used on three design projects, and I compare this performance to that of other programming environments available to architects.

7.6 Benchmarking Yeti

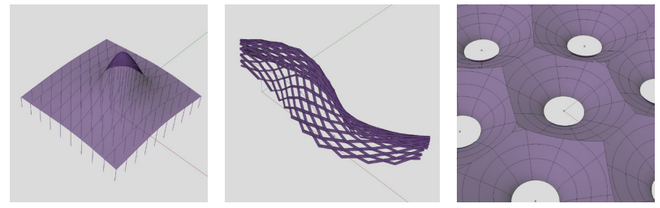


Figure 68: The three benchmark projects in Yeti. Left to right: Kilian's first roof, Kilian's second roof, and the hyperboloids from the Responsive Acoustic Surface.

Method

To test the viability of generating parametric models with interactive programming, I carried out three design tasks using Yeti (fig. 68). It was not clear whether interactive textual programming would cope with creating a parametric model, let alone creating one in the midst of an architecture project. I therefore selected three design tasks that stressed a number of essential parametric techniques while letting me clearly isolate and observe the performance of interactive programming. The first two design tasks come from a pair of tutorials Axel Kilian developed in 2005. The tutorials teach designers to model a pair of parametric roofs and introduce "several key parametric modelling concepts" (Woodbury, Aish, and Kilian 2007, 226) such as arrays, constraints, and instantiation. Recreating the tutorials in Yeti ensures these key parametric modelling concepts are also possible with interactive programming. The third design task revisits the plaster hyperboloids of the Responsive Acoustic Surface. Given the computational challenges in calculating the intersections between hyperboloids, the project is an ideal setting for finding the limits of Yeti's interactivity.

As a benchmark I repeated the three design tasks with two established methods of programming, both of which I am adept at: interactive visual programming in Grasshopper (version 0.8.0052), and textual programming with Rhino Python (in Rhino 5, version 2011-11-08). By repeating the design tasks I was able to compare Yeti's performance to that of established programming methods through the metrics established in chapter 4. In particular I was interested in the following qualitative metrics:

- **Functionality:** Are all the modelling tasks able to be performed by every programming method?
- **Correctness:** Do programs do what is expected?
- **Ease of use:** Are the modelling interfaces easy to use?

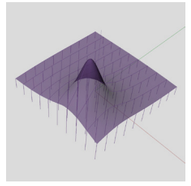
I was also interested in the following quantitative metrics from chapter 4.3:

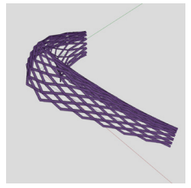
- **Construction time:** How long did the respective models take to build?
- **Lines of Code:** How verbose were the various programming methods?
- **Latency:** How quickly did code changes become geometry?

The intention is not to definitively say one programming method is better than the other, rather the intention is to capture the primary differences between these programming methods while verifying that Yeti can complete the same key design tasks. Recent studies employing a similar method include: Janssen and Chen's (2011) comparison of the visual programming environments Grasshopper, GenerativeComponents, and Houdini; Leitão, Santos, and Lopes's (2012) comparison of Grasshopper and the textual language Rosetta; and Celani and Vaz's (2012) comparison of Grasshopper and the textual language Visual Basic. The first hand accounts in these studies are largely successful at establishing the primary differences between the programming methods they compare, differences I aim to establish in this case study of interactive textual programming.

Benchmark 1 and 2: Kilian Roofs

When Axel Kilian developed his pair of parametric modelling tutorials in 2005, neither Grasshopper nor Rhino Python had been invented and GenerativeComponents was still two years away from being commercially available.[5] For architects who had never encountered parametric modelling, Kilian's tutorials showcased "several key parametric modelling concepts and quickly yielded a form with some architectural credibility" (Woodbury, Aish, and Kilian 2007, 226). In particular, each tutorial teaches students how to model a roof that adapts to its context, while also introducing students to dataflows, arrays, b-splines, and the instantiation of objects that are topologically identical but physically different. To complete these tutorials in Grasshopper, Rhino Python, and Yeti, all the programming methods must be capable of performing the essential parametric modelling techniques outlined in Kilian's tutorials.





5 It is remarkable to consider how much parametric modelling has changed in the seven years since Kilian's tutorials, both in terms of the number of architects using parametric models and in terms of range of parametric modelling environments available to architects. While Kilian's tutorials are just seven years old, in many respects they have an historic credence through which it is possible to track the development of parametric modelling.

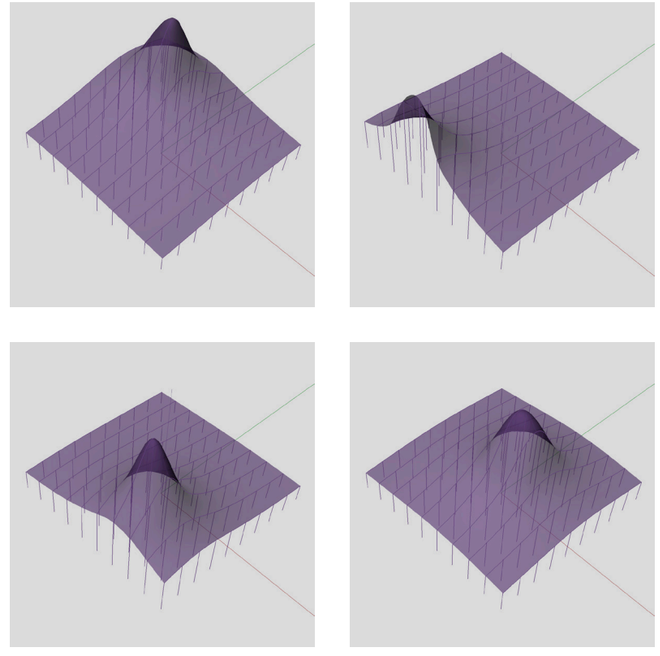
Functionality and Correctness

Both of Kilian's roofs could be recreated in Grasshopper, Python, and Yeti. In this sense all the environments were correct: the code from every modelling environment correctly generated the expected geometry. There are however differences in functionality. Yeti has a limited geometric vocabulary in comparison to either Grasshopper or Python. While this was not a hindrance in creating the roof models, on other projects this may prevent Yeti from correctly generating the required geometry (at least until Yeti's vocabulary is further developed). In this sense Grasshopper and Python are more functional than Yeti since they both offer what Meyer (1997, 12) calls, a far greater "extent of possibilities provided by a system." Beyond the geometry of the various modelling environments, there are a number of key differences in functionality that I will expand upon shortly, including the management of lists, the baking of geometry, and the creation of custom objects.

Construction Time

The first roof (fig. 69) took me four minutes to build in Grasshopper, six minutes to build in Yeti and sixteen minutes to build in Python. I recorded

Figure 69: Four variations of Kilian's first roof generated with Yeti. The roof rests on a grid of columns whose height varies to accommodate a tree under the roof. The height of any particular column is a function of the distance between the column and a point representing the tree. When the point moves, the roof readjusts to allow for the tree's new location.



myself building the Python model and in the video it is clear that the roof took a while to build because I spent a lot of time writing code to manage arrays and bake geometry. Since I had no feedback about whether my code worked, I then had to spend time testing the Python code by cycling through the Edit-Compile-Run loop. Both Grasshopper and Yeti have built in support for simple arrays and geometry baking so I did not have to spend time creating them, which led to simpler models and a reduced construction time.

The second roof (fig. 70) is geometrically more complicated than the first. In Grasshopper the model took twenty-five minutes to build, involving many manipulations of the standard array to generate the diagonal pattern. These array manipulations were less of a problem in Yeti and Python since both environments allowed me to define a diagonal panel that could then be instantiated across a surface. Because of the way geometry is generated relative to axes in Yeti and Python, modelling the roof's parabolic ribs and aligning them to the path was surprisingly difficult in both programming environments. In the end the model took forty minutes in Yeti and sixty-five minutes in Python.

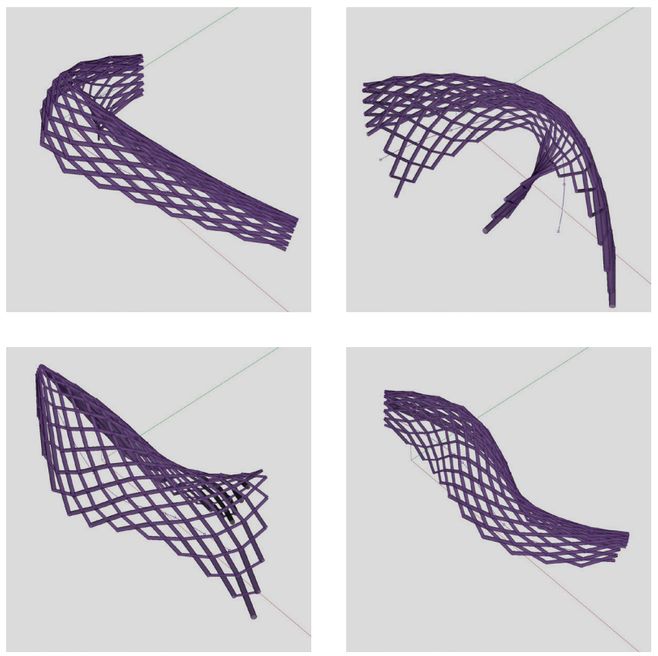


Figure 70: Four variations of Kilian's second roof generated with Yeti. The roof is made an array of parabolas lofted together to make a surface that is then diagonally crisscrossed with tubes. The parabolas follow the path of a curve and if the curve is adjusted, the roof readjusts to follow the curve.

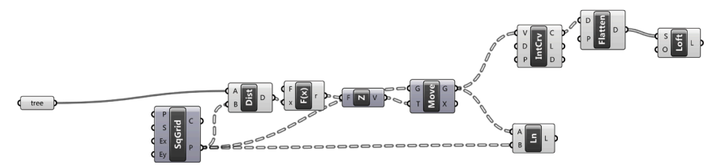
Lines of Code

The Yeti scripts and the Python scripts were of a similar length (fig. 71); the first model required thirty-six lines of code in Yeti and thirty-five in Python, while the second model required ninety-three lines of code in Yeti and seventy-eight in Python. Although the programs have a similar number of lines, there are very few correlations between lines due to the differences between the two programming paradigms (Yeti being dataflow based and Python being object-oriented). The Yeti code is noticeably

Figure 71: The roof from Kilian's first tutorial in Yeti (left), Python (right), and Grasshopper (bottom). While the Yeti and Python code are of a similar length, the lines of code do not correspond due to the differences in programming paradigms. The Yeti code is also noticeably sparser than the Python code. But both the Python and Yeti code looks verbose when compared to the equivalent code in Grasshopper.

```
row: &rows
  treeLoc:
    x: 1
  xLoc:
    from: -5
    to: 5

loft: &roof
  addprofiles: *rows.splines


--- !row
unit: &rowOfUnits
  treeLoc: !treeLoc
  unitLoc:
    x: !xLoc
    y:
      from: -5
      to: 5
    visible: 0

spline: !splines
  addpoints: *rowOfUnits.colu
...

--- !unit
point: !treeLoc
point: !unitLoc
  visible: 0

vector: &toTree
  start: *unitLoc
  end: *treeLoc
  visible: 0

double: &height (10/(*toTree

line: !column
  start: *unitLoc
  direction:
    x: 0
    y: 0
    z: *height
...
```

```
class column:
  def __init__(self, location, tree):
    distance = rs.Distance(location, tree);
    height = 10 / (distance + 1)
    self.topPt = Rhino.Geometry.Point3d(locatic
    self.topPt.Z = height;
    self.line = Rhino.Geometry.Line(location, s

  def getTopPt (self):
    return self.topPt

  def draw(self, doc):
    doc.Objects.AddLine(self.line)

class columnRow:
  def __init__(self, x, tree):
    self.columns = list()
    pts = list()
    for i in range(10):
      self.columns.append(column(Rhino.Geometry
      pts.append(self.columns[i].getTopPt())
    self.curve = rs.AddInterpCurve(pts)

  def getProfileGUID(self) :
    return self.curve

  def draw(self, doc):
    for col in self.columns:
      col.draw(doc)

tree = Rhino.Geometry.Point3d(0, 0, 0)

rows = list()
profiles = list()
for i in range(10):
  rows.append(columnRow(i, tree))
  profiles.append(rows[i].getProfileGUID())

rs.AddLoftSrf(profiles)

doc = Rhino.RhinoDoc.ActiveDoc

for row in rows:
  row.draw(doc)

doc.Objects.AddPoint(tree)
```



sparser than the Python code and contains on average just ten characters per line, whereas the Python code contains twenty-five characters per line. This is predominantly because Python is a general-purpose language, so differentiating commands generally requires more verbosity than in Yeti (for example, the point command in Python requires twenty-two characters [`Rhino.Geometry.Point3d`] whereas in Yeti it requires just six characters [`point:`]).

The size of a visual program is not directly comparable to the size of a textual program but, having said that, the Grasshopper model for the first roof does look smaller and less complex than the corresponding textual programs (fig. 71). The Grasshopper model for the first roof contains just ten nodes and has a cyclomatic complexity of five, which means it is about half the size of the median Grasshopper model (see chap. 4.3). In comparison, the Grasshopper model for the second roof contains fifty-two nodes and has a cyclomatic complexity of twenty-four (fig. 72).

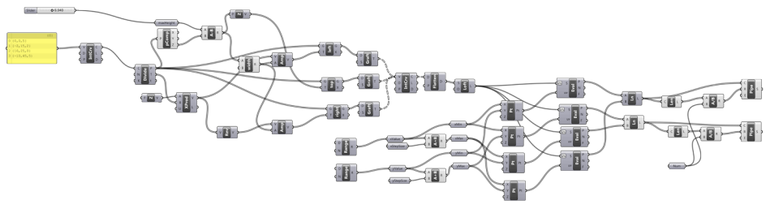


Figure 72: The roof from Kilian's second tutorial in Grasshopper. The lack of structure in this model makes it difficult to understand the model's fifty-two nodes.

The second model begins to exhibit some of the problems typical of larger unstructured visual programs that I discussed in chapter 6. In particular, it is almost impossible to infer the model's function by just looking at the nodes, and even knowing the model's function, it is difficult to do things like identify the nodes that generate the roof shape or understand why four nodes generate points just past midway in the model. While the code for the Yeti and Python models can also be hard to understand, the structure inherent to textual programs at least provides a few clues to aid reading the models.

Latency

Yeti remained interactive while designing both of the roofs. On the first roof, code changes took on average 50ms to manifest in changes to the geometry. On the second roof these changes took 27ms. Grasshopper was similarly responsive, implementing changes on the first model in 8ms and taking 78ms on the second model. All of these response times fall well inside Miller's threshold of 100ms, which is the threshold for a system to feel immediately responsive (Miller 1968, 271; Card, Robertson, and Mackinlay 1991, 185). Python fell outside this threshold, taking 380ms to generate the first model and 180ms to generate the second. These times only measure the running time of the Python program and do not include the time the spent activating and waiting for the Edit-Compile-Run loop. When these other activities are taken into consideration, the Python code takes on average between one and two seconds to execute.

Ease of Use

Ease of use is hard to define since it depends on the "various levels of expertise of potential users" (Meyer 1997, 11). The Kilian models demonstrate that, at the very least, interactive textual programming in Yeti can match the functionality both of non-interactive textual programming in Python, and of interactive visual programming in Grasshopper. These functional similarities, combined with similarities in code length and slight improvements in construction time, indicate that interactive programming is a viable way to textually program parametric models. The reductions in latency are apparent when reviewing the videos of the various models being created. In the videos of Grasshopper and Yeti, the geometry is continuously present and changing in conjunction with the code. The distinction that often exists between making a parametric model (writing the code) and using a parametric model (running and changing the code) essentially ceases to exist in Yeti since the model is both created and modified through the code: toolmaking and tool use are one and the same. However, it remains to be seen whether the interactivity borne of a reduced latency improves the ease of use independent of the any particular user's expertise.

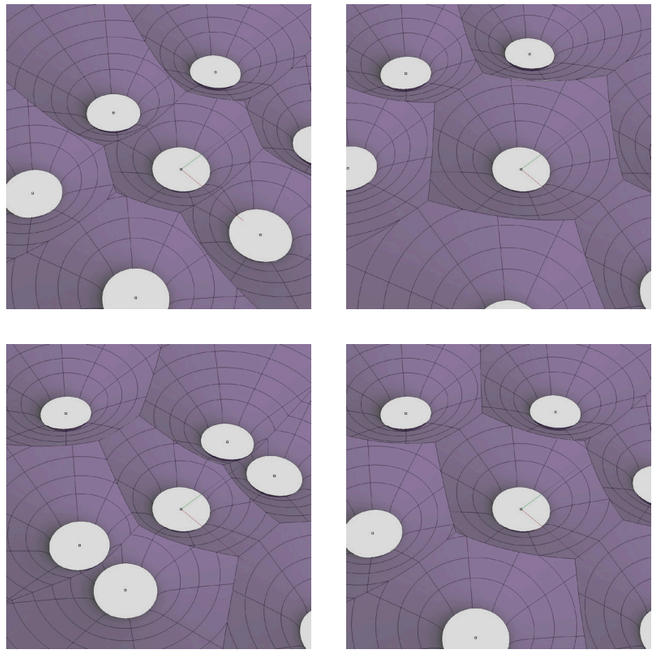


Figure 73: Four variations of the Responsive Acoustic Surface's hyperboloid layout generated with Yeti. Slight changes in the hyperboloid position significantly alter the shape of the bricks.

Benchmark 3: SmartGeometry Redux

In preparation for SmartGeometry 2011, the project team experimented with creating the Responsive Acoustic Surface in a variety of parametric modelling environments: Grasshopper, Digital Project, and Open Cascade. To best utilise the relative strengths of the various modelling environments, the hyperboloid brick was developed in a workflow that threaded the design between Grasshopper and Digital Project. Changes in this workflow took many minutes to propagate due to the time taken in exchanging data between software and the time taken in finding the intersections between hyperboloids. There was minimal feedback during this process and, as a result, the final relationship between hyperboloids was not obvious. The relationship would only become obvious when we built the hyperboloids, stacked them, and realised they did not quite fit together. The hyperboloids' fit comes down to subtle nuances in the planarity of the intersections. Given the difficulty of calculating these intersections, the Responsive Acoustic Surface challenges all varieties of parametric model. By repeating the project with Yeti, the intention was to see if Yeti could remain interactive on such a challenging project and to see if the interactivity helped to understand the design better prior to construction.

Creating an array of hyperboloids was relatively straightforward in Yeti and not substantively different to distributing panels over the roof in Kilian's second tutorial. The challenging part was intersecting the hyperboloids and then deciding which part of hyperboloids to keep. In a procedural paradigm this is easily expressed with an `if-then-else` structure:[6] *if* part of the hyperboloid is past the intersection plane *then* delete the part *else* keep the part. The `if-then-else` structure is not yet included in Yeti primarily because adding it does not seem consistent with the rest of Yeti's syntax. As a temporary workaround, the logic for deciding which part of the hyperboloid to keep was expressed procedurally in Yeti rather than expressed in Yeti's YAML code. These challenges indicate some important functional differences between the dataflow paradigm of Yeti and the procedural paradigm of other textual languages.

The hyperboloid intersections were too arduous to calculate in real time with either Grasshopper or Yeti. The project could only be completed by pausing the interactivity, which effectively reverted Yeti back to the manual Edit-Compile-Run loop. Being able to revert to this non-interactive paradigm was useful to grind out the computationally taxing geometry of the hyperboloids, but reverting to a non-interactive paradigm also removes the primary impetus for creating Yeti in the first place. So while interactive textual programming is useful for straightforward calculations, on computationally difficult projects the Edit-Compile-Run loop may be inescapable, which possibly makes errors, like those contained in the original hyperboloid bricks, unavoidable.

6 The if-then-else structure is one of the three Böhm and Jacopini (1966) identified. They denote it with the symbol ∏. See chap. 6.2.

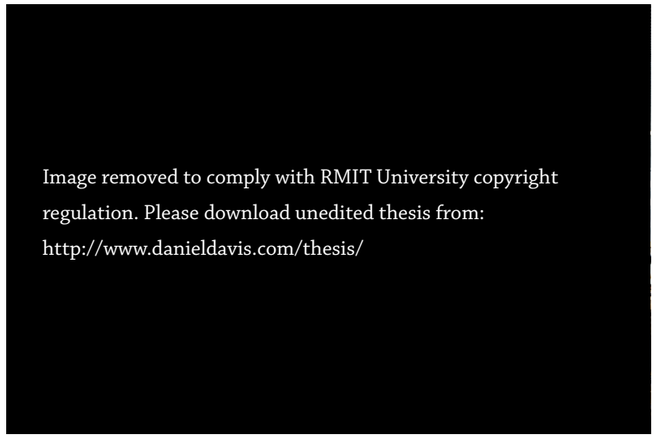


Figure 74: The entrance to the FabPod, situated within the RMIT DesignHub, Melbourne (March 2013).

SmartGeometry Redux Redux: Fabpod

A second iteration of the Responsive Acoustic Surface was developed as part of a design studio Nick Williams and John Cherrey ran at RMIT University in 2012 (with assistance from a research team led by Jane Burry and Mark Burry). The studio considered how the hyperboloid bricks of the Responsive Acoustic Surface could be adapted to acoustically diffuse sound in a meeting room. During the studio, the students designed a variety of meeting rooms for an open-plan office and later constructed one of the designs, which was dubbed the FabPod.

Based on analysis I had done for the Responsive Acoustic Surface, it was known that the hyperboloid bricks would only enclose spherical volumes.[7] Previous research by Brady Peters and Tobias Olesen (2010) had suggested that the best acoustic performance would come from non-periodic tilings of the hyperboloids. For the FabPod, this was achieved by distributing the hyperboloids irregularly across spherical surfaces, and then trimming each hyperboloid where it intersected its neighbours. Doing so required

7 The bricks have a timber frame supporting the edges of the hyperboloids. Since it was only practical to build the frame from planar sections, the edges of the hyperboloids had to be planar as well. My analysis for the Responsive Acoustic Surface had demonstrated that adjoining hyperboloids only had planar edges in a limited range of circumstances: (1) the adjoining hyperboloids had to be of the same size, (2) the normals had to either be parallel or converge at a point equidistant from the hyperboloids. This can only occur if the hyperboloids are distributed on a planar surface with the normals parallel to the surface normal, or on a spherical surface with the normals pointing towards the centre. The FabPod uses spherical surfaces since they were shown to have better acoustic properties.

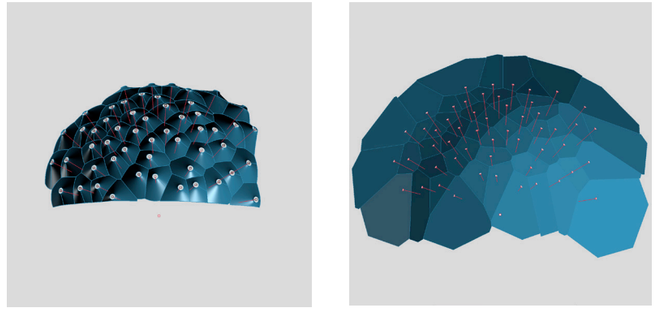


Figure 75: Left: An early study of hyperboloid intersections that I produced in January 2011 for the Responsive Acoustic Surface. The model proves that hyperboloids distributed on a spherical surface intersect with planar curves. Right: The intersections between hyperboloids also form a Voronoi pattern; shown is the output from the FabPod's spherical Voronoi parametric model.

finding the intersections between 180 hyperboloids, which was vastly more complicated than finding the intersections between the 29 regularly distributed hyperboloids of the Responsive Acoustic Surface. Further adding to the difficulty, the intersections were needed not only for creating the construction documentation at the end of the project, but also for generating models accurate enough to run the acoustic simulations used regularly throughout the project. Given how often these intersections were needed, I once again considered whether this process could be made interactive.

I began re-examining how the hyperboloid intersections were being generated. In previous parametric models (including the Yeti model) the hyperboloids were represented as NURBs surfaces and the intersections were calculated using numeric algorithms. While there are various numeric algorithms for finding the intersections between NURBs surfaces, in essence, all the algorithms involve iteratively moving a curve along one surface until the curve lies within a specified tolerance of the other surface (Patrikalakis 1993). Analytic equations are an alternative to using numeric algorithms. An analytic equation derives directly from a surface's mathematical formula, which allows the intersection curve to be generated by directly solving the equation rather than spending computational resources doing iterative calculations. While analytic equations have some potential efficiencies, prior to this research, there was no existing analytic equation for calculating hyperboloid intersections.

Generating the analytic equation for the hyperboloid intersections was a multi-stage process. I first proved that the intersection planes between hyperboloids correspond with the pattern from a spherical Voronoi algorithm I developed (fig. 75). Lines between the Voronoi pattern and the sphere centre always correlate with points on the hyperboloid intersection curves. I derived an analytic algorithm to find these points by taking the formula for a hyperboloid:

$$\frac{x^2}{a^2}+\frac{y^2}{a^2}-\frac{z^2}{c^2}=1$$

And the formula for a line:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} i \\ m \\ p \end{bmatrix} + \begin{bmatrix} j \\ n \\ q \end{bmatrix} \cdot t$$

Substituting to eliminate *x*, *y* & *z*:

$$\frac{(i+jt)^2}{a^2} + \frac{(m+nt)^2}{a^2} - \frac{(p+qt)^2}{c^2} = 1$$

Which rearranges to give the value of *t* from the original line formula:

$$t = \frac{-2 \cdot (c^2ij + c^2mn - a^2pq) \pm \sqrt{(2c^2ij + 2c^2mn - 2a^2pq)^2 - 4 \cdot (c^2j^2 + c^2n^2 - a^2q^2) \cdot (c^2 \cdot (i^2 + m^2) - a^2 \cdot (p^2 + c^2) - 1)}}{2c^2(j^2 + n^2) - 2a^2q^2}$$

Using this analytic equation I developed a parametric model in Grasshopper for calculating the FabPod's hyperboloid intersections. In the previous Grasshopper and Yeti models, calculating the intersections between 180 hyperboloids took approximately two and a half minutes (150,000ms). By utilising the analytic equation I was able to generate the same intersections in 250ms, which is one six-hundredth of the previous times and fast enough to feel interactive. With the intersections calculated so quickly, students in the workshop were able to make many

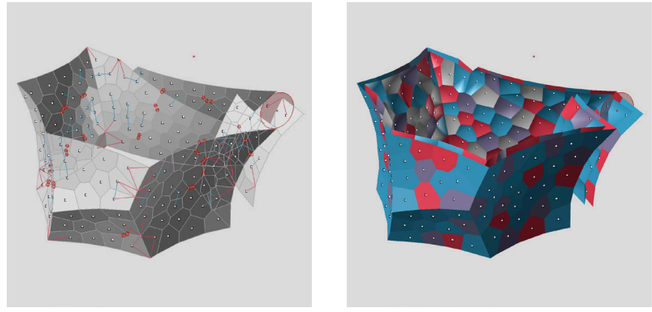


Figure 76: Left: The final spherical Voronoi pattern used in the FabPod. The blue and red lines provide feedback about the spacing of the hyperboloids and their constructability. Right: The final parametric model of the FabPod's hyperboloid intersections. The colours correspond with construction materials.

small adjustments to their hyperboloid layouts while receiving real-time feedback about potential construction problems (edges that were too short, and hyperboloids that were too close or too far apart; fig. 76). All of these problems had to be eliminated in order for the FabPod to be constructible. I experimented with using hill-climbing and dynamic relaxation to remove the problems, but the search space was too disjointed to make this type of optimisation viable. Therefore the only way to ensure the FabPod's constructability was to move each hyperboloid until the construction problems were resolved. If students had to wait two and a half minutes to see the outcome of every movement this would have been an unbearable task, which makes the real-time feedback supplied by the analytic algorithm an essential component in the FabPod's viability.

Typically software engineers caution against spending large amounts of time optimising algorithms to reduce latency. Bertrand Meyer (1997, 9) warns, "extreme optimizations may make the software so specialized as to be unfit for change and reuse." This is certainly true of my analytic algorithm, which is so highly tuned to calculating the FabPod's hyperboloid intersections that it is of little use to any other project. On the other-hand, the generalised optimisations of Yeti are applicable in a wide range of circumstances, but not powerful enough to ensure the viability of the FabPod. In reducing parametric model latency there is a balance to find between extendability, correctness, reusability; a balance activated by the architect's ability to explore multiple ways of generating parametric models.

Figure 77: Panorama of FabPod in the final stages of construction at the RMIT DesignHub, Melbourne (February 2013).

7.7 Conclusion

Unlike at SmartGeometry 2011, I was not up at 3 a.m. writing code in the hours before the start of the FabPod workshop. Even more thankfully, there were no undetected errors lurking in the hyperboloid bricks and the project was constructed largely without incident. There are many reasons for this improvement: we knew the geometry better, we had a better construction system, the project was better managed, and we had better feedback while we were designing. Rather than blindly typing code and hoping (as we had done at SmartGeometry 2011) that the code output was correct, at the FabPod workshop we had immediate feedback regarding potential construction errors.

Immediate feedback has not always been possible for architects developing parametric models. Historically, geometric designers had to make a choice: either use an interactive visual editor, accepting the problems of scale this raises (see chap. 6); or forgo interactivity in favour of writing the code with text. Many people, including Ivan Sutherland (1963, 8), Bret Victor (2012), and Nigel Cross (2011, 11), have suggested that forgoing interactivity is undesirable since feedback is a vital part of the design process and one best delivered immediately. Their intuition is backed up by cognitive studies that show that novice programmers need progressive feedback (Green and Petre 1996, 8), and that designers suffer from change blindness when feedback is delayed (Erhan et al. 2009; Nasirova et al. 2011; see chap. 2.3). In other design disciplines, designers have access to a range of interactive textual programming environments yet, for architects, interaction and textual programming were incompatible prior to my research.

In this chapter I have demonstrated how Yeti's novel method of interactive textual programming supports architects designing geometry. Unlike existing methods of interactive programming – which are ill equipped to accommodate the computational intensity of geometric calculations – Yeti enables the interactive creation of geometry by using a Directed Acyclic Graph (DAG) to manage code changes. In order to generate the DAG, Yeti is based on the relational markup language YAML, which is paradigmatically different to procedural programming languages but comparable in terms of construction time, code length, and functionality. Unlike many procedural programming environments, Yeti also incorporates a number of innovations software engineers have developed to make the Edit-Compile-Run loop feel more interactive, such as real-time error checking, autocompletion, and interactive debugging.

By using YAML to create a DAG, Yeti is able to reduce the latency between writing code and seeing the geometric results. On certain projects, like Kilian's two roofs, the reduction in latency transforms a task that designers would typically do without any feedback into one designers can do with constant feedback. As a result, writing code and modifying code in Yeti become one and the same. On other projects, like the hyperboloids of the Responsive Acoustic Surface, Yeti does not reduce the latency sufficiency for interaction to occur and Yeti has to fall back on the Edit-Compile-Run loop. However, the FabPod demonstrates that designers can further reduce latency by trading off extendability, correctness, and reusability. In the case of the FabPod, this reduction in latency made a significant contribution towards identifying and then eliminating any construction problems. This indicates that qualities of a parametric model's flexibility – like the model's latency – can have a discernible impact on a project's design. These qualities can themselves be designed through the composition of the parametric model or through the selection of the programming environment. Yeti demonstrates how knowledge from software engineering can offer a pathway towards more diverse programming environments that can be tuned for particular attributes of parametric modelling. Yeti is just one manifestation of this knowledge and there are many more possibilities that make programming environments for architects an obvious location for future development.

8 Discussion: Beyond Toolmaking

Currently "little explicit connection" exists between the practice of parametric modelling and the practice of software engineering, writes Robert Woodbury (2010, 66). In my research I have sought to establish such connections by exploring whether the design of software can inform the design of flexible parametric models. More specifically, I have taken three concepts from the *Software Engineering Body of Knowledge 1.0* (Hilburn et al. 1999) and observed, using a reflective practice methodology, their affect when applied to the parametric models of various architecture projects. In the following pages I reflect upon what these case studies contribute to our understanding of parametric modelling and, in particular, our understanding of parametric modelling's relationship to software engineering. I argue there are connections between software engineering and parametric modelling centred around shared challenges, shared research methods, and shared design practices. These connections position software engineering as an important precedent for architects; a relationship that has implications for how parametric modelling is taught, for how parametric modelling is integrated in practice, and for how we conceive of parametric modelling.

8.1 Shared Challenges

The challenges with parametric modelling are rarely reported, although they easily experienced. Thomas Fischer (2008, 245) concludes his doctoral thesis by lamenting that firsthand accounts of "failures and dead-ends … seem to be rare and overshadowed by the great number of post-rationalised, outcome-focused reports on digital design toolmaking." Against this backdrop, one contribution of my research has been to collate the fragmented reports of parametric modelling failures (see chap. 2.3). Sometimes these reports are just a single offhand sentence tucked into a five hundred-page thesis revealing the unnervingly catastrophic behaviour that if the "topology of a project changes the [parametric] model generally needs to be remade" (Gerber 2007, 205). Sometimes these reports come from experts with decades of parametric modelling experience, which inspires them to write tell-all papers about changes breaking models, about a lack of reuse, and about changes having unintended consequences (Smith 2007). These fragmented reports collectively signal that the networks of explicit functions underlying parametric models are vulnerable to being broken by the very thing they are designed to accommodate: change (see chap. 2.3). In many cases the complexity of the parametric relationships leave the designer with only two choices: delay the project and rebuild the model, or avoid the change all together and accept an outcome that was not so much created with a parametric model as much as it was created for a parametric model. This is a challenge often encountered but rarely published.

Software engineers face similar challenges (see chap. 3.1). Like architects creating parametric models, software engineers need to express outcomes in logically precise instructions for the computer. These instructions are susceptible to being broken as the outcomes of the project inevitably change with the project's development. For a period in the 1960s, scientists feared the breakages would be insurmountable and the limits of computation would not be computer speed but rather the cognition of the programmers creating and maintaining software (Naur and Randell 1968, chap. 7.1). The challenges of 1960s *software crisis* gave rise to the discipline of software engineering (see chap. 3.1). These are challenges that software engineers have been grappling with for decades, challenges that resemble the fragmented reports of parametric modelling failures.

An important caveat is that creating software is similar, but not identical, to creating architecture. Broadly speaking, parametric models have a very particular user (often the model's developer or colleague), product (typically the product is the architecture rather than the model), team size (normally just a few people), and project lifetime (often measured in months) (see chapter 3.2 for more details). None of these idiosyncrasies are necessarily abnormal in the context of software engineering, but they are not necessarily common either. This suggests that not all of the challenges faced by software engineers are equally relevant to architects. For instance, architects are likely to have more in common with the challenges faced by a small team of software engineers delivering a project on a tight schedule than they are with the challenges faced by a large team of software engineers developing an operating system to last many years. With this caveat in place, there are many commonalities between the challenges of architects and software engineers.

In some respects the commonalities are unsurprising. A parametric model is, after all, simply a type of algorithm (see chap. 2.1; Dino 2012). Even as far back as 1993, reports were surfacing that parametric modelling was "more similar to programming than to conventional design" (Weisberg 2008, 16.12). Given the known "common ground" (Woodbury 2010, 66) between the two practices, the surprise is that almost no literature connects the struggles of architects with the struggles of software engineers (Woodbury being one exception but even within his writing this connection is only tangentially explained). My research suggests that the challenges architects using parametric models encounter with change are shared to some degree by software engineers, a connection that has implications for how architects may address these challenges.

8.2 Shared Methods

The flexibility of a parametric model is often framed in a binary of failure and success. My research suggests flexibility is far more nuanced. Parametric models appear to have multiple types of flexibility that are traded off against one another through modelling decisions. To articulate these flexibilities in the case studies I have drawn upon the vocabulary software engineers use to describe qualities of computer code. The case studies show the applicability of quantitative descriptions like lines of code, cyclomatic complexity, construction time, modification time, latency, and dimensionality (see chap. 4.3), as well as Bertrand Meyer's (1997, chap. 1) qualitative descriptions of correctness, robustness, extendability, reusability, compatibility, efficiency, portability, ease of use, functionality, and timelessness (see chap. 4.4). While these sixteen metrics appear to give a relatively full picture of flexibility, there is certainly scope for further connections between shared methods of appraising software engineering and parametric modelling.

The vocabulary of software metrics helps articulate the differences between various parametric models in my research. By quantitatively measuring 2002 parametric models generated by 575 designers I have been able to show that model size and cyclomatic complexity are strongly correlated, just like they are in software engineering (see chap. 4.3). The survey also established that the average Grasshopper model contains twenty-three nodes, with a high cyclomatic complexity, and virtually no structure (see chap. 4.3 & 6.3). This is first time a large collection of architectural parametric models has been analysed, and it is perhaps the first description of parametric modelling not reliant upon firsthand accounts. In the case studies I was able to combine the quantitative and qualitative metrics to triangulate a more comprehensive understanding of each parametric model's flexiblity. For example, in the interactive programming case study (chap. 7) it was shown that Yeti's impact on model latency also had implications for the construction time, functionality, ease of use, and correctness of the model. Being able to describe the flexibility of a parametric model using a vocabulary more nuanced than the current binary of failure and success is a potentially important contribution.

It is important to caution that these measurements are not necessarily predictors of model behaviour. A model may have a low cyclomatic complexity and low latency, be robust and easy to use, but still break with an unanticipated change. Another model, a model that looks to be in far worse condition, may go through the same changes effortlessly. In a similar vein, these metrics are unlikely to measure successfully a project's progress or quality. Attempts to manage programmers using similar metrics have never been widely successful, which has led one prominent advocate of software engineering metrics, Tom DeMarco, to recently say:

> My early metrics book, *Controlling Software Projects: Management, Measurement, and Estimation* (1982), played a role in the way many budding software engineers quantified work and planned their projects. In my reflective mood, I'm wondering, was its advice correct at the time, is it still relevant, and do I still believe that metrics are a must for any successful software development effort? My answers are no, no, and no.
> *Tom DeMarco 2009, 96*

This is a significant retraction from a software engineer perhaps best known for coining the adage, "you can't control what you can't measure" (DeMarco 1982, 3). Although this adage may not ring true, and although software metrics may not be useful in predicting parametric model behaviour, they are still a valuable vocabulary for researchers describing what a model has done.

In applying these various metrics I have built upon the research methods shared between software engineering and architecture. These methods are already connected to some degree, since software engineers and architects cite common sources like Schön and Cross in their research design. The connection has been further bolstered in recent years by software engineers adopting lean development methods that "sound much like design" (Woodbury 2010, 66), and by attempts to position software engineering (and its associated research) within the field of design (Brooks 2010). While software engineers have shown a willingness to learn from design research, this has largely been an asymmetric exchange. My research has gone against the prevailing by drawing upon software engineering research methods and methodologies to structure research about architecture. While the case studies have shown the potential of this exchange, there remains considerable scope to establish further connections between shared methods of research in software engineering and architecture.

8.3 Shared Practices

The challenges that change presents for both software engineers and architects have pushed both to improve their practices. The progress of one has been largely unbeknownst to the other, which is perhaps most evocatively illustrated in the parallel curves drawn simultaneously by architect Boyd Paulson (1976) and software engineer Barry Boehm (1976) (see chap. 2.2 & 3.1). Both demonstrate, neither aware of the other, that changes become exponentially more expensive as a project progresses. This cost has seen both architects and software engineers attempt to avoid making changes by employing a practice known as *front-loading*. In the decades since Paulson first drew his curve, architects have regularly rehashed the curve and its conclusions to justify the practice of front-loading (Patrick MacLeamy is almost always misattributed as the originator, see chap. 2.2). At the same time, in the decades since Boehm first drew his curve, the practice of software engineering has improved to the point where some commentators have suggested cost now approaches a horizontal rather than vertical asymptote. This is a practice that lets software engineers "embrace change" (Beck 1999) rather than avoiding change with front-loading.

The *Software Engineering Body of Knowledge Version 1.0* (SWEBOK.1999) (Hilburn et al. 1999) attempts to catalogue the knowledge of a software engineer after three years of practice. In my thesis I have hypothesised that aspects of this body of knowledge are applicable not only to the practice of software engineering but also the to the practice of parametric modelling (see chap. 3.2). In my three case studies I have considered how three aspects of the SWEBOK.1999 – programming paradigms, programming structure, and programming environments – affect the practice of parametric modelling:

- In Case Study A (chapter 5) I considered how programming paradigms impacted the creation of parametric models for the Sagrada Família. I developed a new method of parametric modelling using logic programming and found this to influence the parametric model's construction time, modification time, latency, and extendability. This case study suggests a model's programming paradigm is a key control point in tuning the model's flexibility.

- In Case Study B (chapter 6) I experimented with changing the structure of Dermoid's parametric models. Despite structure being a fundamental part of software engineering, the overwhelming majority of the 2002 parametric models I surveyed had no structure. The restructuring of Dermoid's models demonstrated that model structure, rather than model size or cyclomatic complexity, is likely the greatest determinant of model understandability. This has implications for model reuse and project continuity, with structure helping support changes late in the Dermoid project.

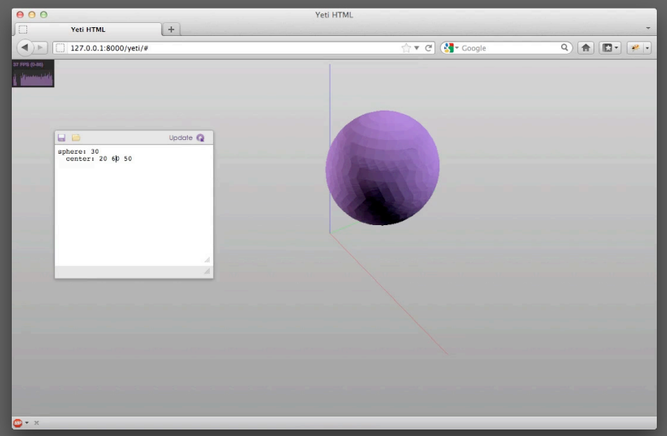
- In Case Study C (chapter 7) I applied innovations from software engineering Integrated Development Environments (IDEs) to create a novel interactive programming environment specifically for the challenges of modelling geometry. Across a series of projects this environment reduced the latency of writing code, which has implications for the change blindness (see chap. 2.3) designers sometimes experience when making changes. This case study suggests that the environments architects use to write programs can themselves be sites of innovation.

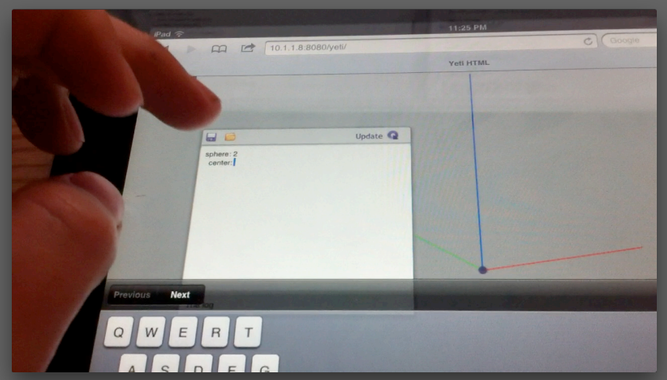
These case studies each individually contribute a novel method of parametric modelling to the field of architectural design (each has been previously published: Davis et al. 2012; Davis et al. 2011a; Davis et al. 2011b; Davis et al. 2011c). I have been able to prototype these new approaches by building upon numerous developments belonging to what the SWEBOK.1999 classifies as *Computing Fundamentals*; developments in programming languages, geometric APIs, operating systems, and computer hardware. Even ten years ago there was so little to build upon that creating any sort of parametric modelling environment (like GenerativeComponents) was considered a major achievement. If this trend continues – and at the moment there is no reason to suspect it will not – over time it should become even easier to test and apply new modelling approaches. For example, I rewrote Yeti (the interactive programming environment from chapter 7) to run upon the newly developed pythonOCC and Django frameworks, with HTML5 WebGL as the rendering engine (which was only a few months old at the time). This rewritten version of Yeti runs on any web browser (fig. 78) and suggests a future where developments in *Computing Fundamentals* empower individuals to rapidly invent novel modelling methods for the peculiarities of a project.

Collectively the three case studies indicate that the software engineering body of knowledge is a fertile ground for improving the practice of parametric modelling. Software engineers have, according to Young and Faulk (2010, 439), spent significant time considering "the essential challenges of complexity and the cost of design" since this is the "primary leverage point" in a discipline where "the costs of materials and fabrication are nil." The case studies demonstrate that the work software engineers have put into addressing these challenges – challenges partly shared by architects – are often applicable to the practice of parametric modelling. Yet, the case studies only begin to explore the application of this knowledge. Even my work on structuring parametric models (chap. 6) only touches the surface of an extremely rich area of software engineering. Likewise, my research into language paradigms and programming environments presents only one instance of many potential possibilities. Other promising sites for future research (identified in chapter 3.2) include the design, coding, and testing stages of *Software Product Engineering* as well as *Software Management* (which is an area of knowledge with many obvious connections to architecture, but one that I have not directly explored in my research).

The software engineering body of knowledge is not the silver bullet to the challenges architects face when working with parametric models. It bears remembering that less than half of all software projects in 2012 were successful (The Standish Group 2012; The Standish Group 2009; Eveleens and Verhoef 2010). What the body of knowledge offers is a precedent for thinking about the practice of parametric modelling. In the case studies these all involved tradeoffs, for example, logic programming (chap. 5) facilitated the un-baking of explicit geometry but also negatively impacted the model's modification time and ease of use (for a detailed reflection on these tradeoffs see the discussions in chapters 5, 6, & 7). Yet within these tradeoffs are options: options to manipulate the flexibility of the parametric model in ways that did not exist before. Potentially the software engineering body of knowledge and the connections my research reveals between shared challenges, shared research methods, and shared design practices offers a precedent for partly controlling a parametric model's flexibly – an act that would have significant implications for the practice of architecture.

Figure 78: The WebGL version of Yeti runs on any computer with a web browser and does not require the user to install any proprietary software like Rhino. Top: Yeti running inside the Chrome browser on a desktop computer. Middle and Bottom: Yeti running inside the Safari browser on an iPad (images taken in December 2011).





8.4 Implications

For Education

The *Software Engineering Body of Knowledge Version 1.0* (Hilburn et al. 1999, 20) not only represents what practicing software engineers know, it also represents what software engineers are taught (see chap. 3.2). If the challenges architects and software engineers face are similar, and if the software engineering body of knowledge suggests practices to alleviate these challenges, the question arises: should architects be taught about software engineering when they learn about parametric modelling?

The teaching of parametric modelling has typically been devoid of reference to the practice of software engineering, which is unsurprising given the lack of connection between parametric modelling and software engineering at other levels. Robert Aish (2005, 12) says he aims to get designers to think algorithmically "without demanding that designers become programmers." Aish (2005, 12) goes on to suggest that designers would benefit from what is almost the antithesis of software engineering: a reduction in the "logical formalism" of parametric models. This ambition comes through in the parametric modelling environments Aish has developed while working at Bentley and Autodesk. When architects learn to use these environments, they are ordinarily taught the "keystroke-by-keystroke instructions to achieve specific tasks" says Robert Woodbury (2010, 8). Consequently, the typical parametric modelling pedagogy follows the practices of teaching non-parametric CAD software much more than it follows the practices of teaching programming and software engineering (see chap. 6.3). Woodbury (2010, 9) argues that the cause (although perhaps it is a consequence) comes from designers being "amateur programmers" and naturally wanting to "leave abstraction, generality and reuse mostly for 'real programmers'."

My research suggests that there may be a danger to teaching parametric modelling without the accompanying background of software engineering. With parametric modelling often simplified to keystroke-by-keystroke sequences, it is perhaps unsurprising that even simple software engineering practices, like naming parameters, are not undertaken in 81% of the 2002 models I examined (see chap. 6.3). Regardless of the cause, the consequence is that these unstructured models are demonstrably incomprehensible to other designers. This may be an acceptable situation if designers are, like Woodbury (2010, 9) characterises them, just quickly creating one-off parametric models "for the task at hand." Yet the reported challenges of making changes to parametric models indicates that many designers are generating and retaining models for more than the immediate task at hand. Designers are developing parametric models that evolve throughout the duration of the project, and designers are frequently using their models to address more than an individual task, often capturing the entire logic of a project within a single parametric model (see chap. 2.3). Each of the case studies in this thesis demonstrates how knowledge of software engineering can help architects through these challenging circumstances. Designers seem ill served by an education that seemingly avoids discussion of these challenges in favour of keystroke-by-keystroke instructions that mimic post-rationally glorified parametric projects. The potential danger in sheltering designers from this knowledge is that rather than making parametric modelling easier, it actually become harder in practice.

How parametric modelling should be taught remains an open question and one deserving of further attention. My research tentatively indicates that designers require some understanding of software engineering to get past the point of making tools that solve isolated tasks. As such, there might be a more nuanced spectrum to the binary Woodbury constructs between amateur and real programmer. Identifying the best way to progress designers along this spectrum is outside the scope of my research, however, I will speculate that the way software engineers are taught may offer another connection to guide the future instruction of designers.

For Practice

Another open question implicated in this research concerns how parametric modelling will impact the practice of architecture. As the practice of software engineering has improved, and as programmers have flattened Boehm's curve into Beck's curve (see chap. 3.1), the process of software engineering has radically changed. Boehm's curve suggested a practice whereby change is avoided through front-loading. Programmers are organised in a rigid hierarchy to push a project through an uncompromising linear sequence of requirements, design, construction, and maintenance (see chap. 3.1). Beck's curve suggests an alternative practice whereby iterations and continual feedback allow developers to "embrace change" (Beck 1999) even late in the project (Brooks 2010). Small teams of programmers self-organise to spiral through stages of planning, acting, and reflecting. The Standish Group's industry survey suggests that these agile processes offer "three times the success rate of the traditional waterfall method [a synonym for front-loading] and a much lower percentage of time and cost overruns" (The Standish Group 2012, 25).

The Dermoid case study from chapter 6 signals how similar manipulations of Paulson and MacLeamy's curve (fig. 79) may impact architectural practice. The Dermoid design process began by exploring both material properties and beam propagation strategies, an exploration that would typically fall into the design development stage of an orthodox design process. The design iterated for over a year, cycling through full-scale prototypes, conceptual parametric models, structural analysis, and design detailing. One of the last decisions to be finalised by the team was the shape of Dermoid, which would ordinarily be a pivotal decision made early in the process (possibly on a napkin). By using parametric models to delay this decision, the design team was able to determine Dermoid's shape at a point where they best understood how the shape would impact the structure, the site, the budget, the construction schedule, and the experience of inhabiting Dermoid. This is essentially the reverse of Paulson and MacLeamy's front-loading: rather than making decisions early in order to avoid the expense of changing them later, in Dermoid the cost of change was lowered to the point where critical decisions could be delayed until they were best understood. Robert Woodbury has hypothesised about such

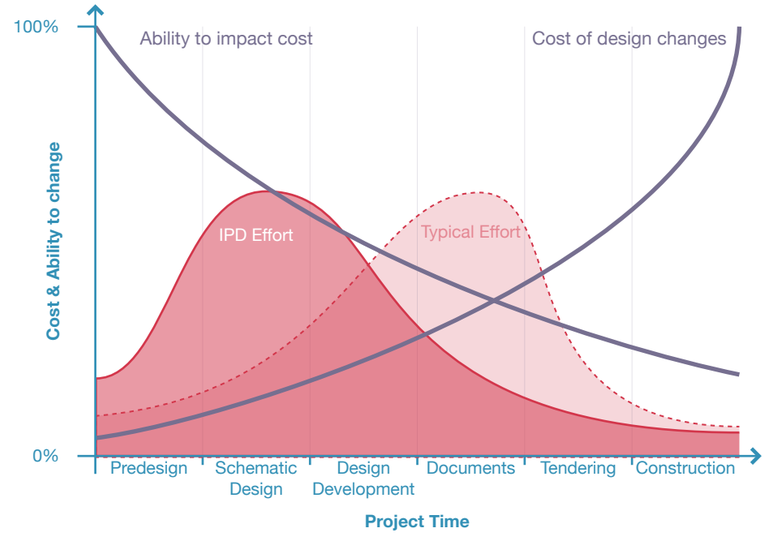


Figure 79: Paulson and MacLeamy's curve (see chap. 2.2). The typical design effort is transferred to an earlier stage of the project – a point where the cost of change is low.

changes to the design process, but provides no examples of this occurring in practice:

> Parametric modelling introduces a new strategy: deferral… Changing the order in which modelling and design decisions can be made is both a major feature of and deliberate strategy for parametric design. Indeed, a principal financial argument for parametric modelling is its touted ability to support rapid change late in the design process.
> *Woodbury 2010, 43*

The parametric models used in the Sagrada Família's frontons, in Dermoid, and in the FabPod all demonstrate how parametric models can accommodate late-stage changes. These changes to the fronton's angle, to Dermoid's shape, and to the FabPod's layout would ordinarily be prohibitively time consuming, but the flexibility of the respective parametric models helped lower the cost of change to the point where the changes were welcomed late in the design process. In contemplating how flexibility may impact the practice of software engineering, Kent Beck asked:

> What would we do if all that investment paid off? What if all that work [improving flexibility] and whatnot actually got somewhere? What if the cost of change didn't rise exponentially over time, but rose much more slowly, eventually reaching an asymptote? What if tomorrow's software engineering professor draws [figure 13] on the board?
> *Beck 1999, 27*

The same questions can be asked of architects. If software engineering techniques enable more flexible parametric models, then perhaps tomorrow's architecture professors will not be drawing Paulson or MacLeamy's curve (fig. 79) on the board but rather a curve that resembles figure 80. My research suggests that the consequence of flattening the cost of change extends beyond financial savings and beyond the ability to make late changes. The real consequence may be a more iterative and malleable design practice; a practice where the positioning of design effort is not dictated by the cost of change but rather by the requirements of the project.

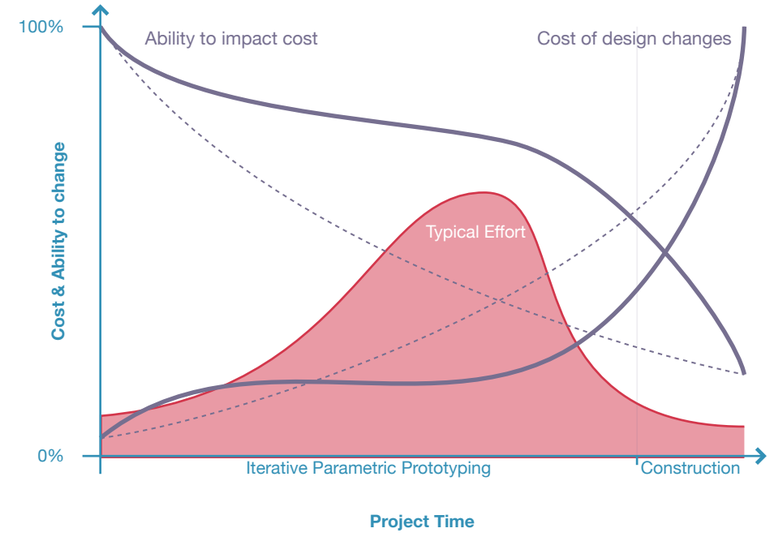


Figure 80: An alternative to Paulson and MacLeamy's curve (shown above). Rather than shifting design effort in relation to the cost of change, it may be possible to shift the cost of change in relation to design effort. My research suggests that parametric models can potentially lower the cost of design changes, allowing designers to defer key decisions until later in the project – by which point they are likely to understand the decision's design consequences better.

Beyond Toolmaking

Edsger Dijkstra, a software engineer I have cited frequently in this thesis, has said of software engineering's relationship to toolmaking:

> Computers are extremely flexible and powerful tools and many feel that their application is changing the face of the earth. I would venture the opinion that as long as we regard them primarily as tools, we might grossly underestimate their significance. Their influence as tools might turn out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity of intellectual challenge.
> *Dijkstra 1970, 7*

Architects have long characterised CAD software as a type of tool; whether it is John Walker in 1983 trying to make "AutoCAD become synonymous with 'drawing tool'" (1983, 1) or whether it is Robert Aish (2011, 23) more recently saying, "software developers do not design buildings. Their role is to design the tools that other creative designers, architects and engineers use to design buildings." Aish goes on to explain the asymmetric relationship borne of "tools transmitting advantage from the toolmaker to the tool user." This relationship between maker and user is disrupted by parametric modelling. As Mark Burry (2011, 8) observes, "digital design is now fully assimilated into design practice, and we are moving rapidly from an era of being aspiring expert users to one of being adept digital toolmakers." He continues, "the tool user (designer) becomes the new toolmaker (software engineer)" (M. Burry 2011, 9 [brackets are Burry's]). This unification of the user and the maker calls into question the distinction between user and maker that has been inherited from other CAD software. To borrow the words of Edsger Dijkstra (1970, 7), by regarding parametric models primarily as tools, we might [have] grossly underestimated their significance.

The distinction between using and making persists in much of the current discourse regarding parametric models. It persists explicitly in the likes of Roland Hudson's (2010) PhD thesis, *Strategies for Parametric Design in Architecture*, when he continually refers to "completed parametric models"

almost as if model making cumulates with a definite point of completion from which tool use and designing can begin (see chap. 2.1). It persists when Robert Woodbury (2010, 9) portrays the parametric model as a tool for adequately *doing* design tasks but never *being* the design task: "the task is foremost, the tool need only be adequate to it" (2010, 9). It persists when Benjamin Aranda and Chris Lasch (2005) write in their book, *Tooling*, that "the job of designing begins" (2005, 9) only once the tool is made. It persists when Patrik Schumacher (2009a, 15) defines parametricism in terms of stylistic outputs coming from "parametric design tools and scripts." And it persists when Mark Gage tells architects to "use computation, but stop fucking talking about it" (2011a, 1) and later tells them to hire software engineers "because these tools are so new to architecture" (2011b, 133). In a less explicit way, the separation between making and use persists in many contemporary definitions of parametric modelling. When authors define *parametric* as being all of design, or only the designs that change, or design in the style of parametricism, they implicitly focus on what parametric models do (see chap. 2.1). By focusing on the doing, many of these definitions overlook the unique features of a parametric model, such as the presence of explicit relationships linking parameters to outcomes; features that distinguish parametric models from traditional manual tools and from other forms of design representation.

"There is something different, unprecedented, and extraordinary about the computer as it compares to traditional manual tools," argues Kostas Terzidis (2006, 24). For Terzidis this difference lies in the inability of humans to reason about computational processes such as parametric modelling. He goes on to remark somewhat cattily in an endnote, "architects such as Neil Denari, Greg Lynn, or Peter Eisenman use the term *tool* to describe computational processes yet none of them has any formal education in computer science" (2006, 34). Indeed, discussions of computer science and software engineering are almost entirely absent from discussions around parametric modelling. Architecture students are generally not taught about software engineering, there is "little explicit connection" (Woodbury 2010, 66) in the academic literature, and many prominent parametric modelling commentators (a number of whom do not themselves use parametric models [M. Burry 2011, 37]) seem more caught up in determining if parametric modelling constitutes a new movement in architecture than they are in acknowledging the real challenges faced by architects using parametric models.

My research has revealed three major connections between parametric modelling and software engineering; connections that link shared challenges, shared research methods, and shared design practices. It is in the shared challenges that the analogy of toolmaking begins to unravel. These challenges are often "overshadowed by the great number of post-rationalised, outcome-focused reports on digital design toolmaking" (Fischer 2008, 245). Yet, in the cracks between the post-rationalised veneer, there are fragments of parametric models that have been shattered by the very thing they were designed to accommodate: change (see chap. 2.3). These catastrophic failures are not from designers quickly creating one-off parametric models "for the task at hand" (Woodbury 2010, 9). Instead, these failures often concern changing the logic of a model that represents an entire project (see chap. 2.3). To borrow a toolmaking analogy, these changes essentially involve turning a tee-square into a french curve while it is being used; a change that is different, unprecedented, and extraordinary compared to any previous drawing tool. Being able to go back and modify a parametric model is a far more distinguishing feature than any outward resemblance to tools in AutoCAD. It is in these modifications that designers are sometimes coming unstuck, but it is also in these modifications that parametric modelling derives its utility and software engineering gains its relevance to the practice of parametric modelling.

This thesis is somewhat unusual in that it chronicles what happens to a range of parametric models throughout a series of projects. These case studies show that design is not something an architect does with a 'completed parametric model', but rather something that happens iteratively throughout the parametric modelling process. They suggest a practice whereby the tool user and toolmaker are indistinguishable, and therefore capable of tuning the model's assorted flexibilities to delay and explore some aspects of the design, while rebuilding sections of the model to accommodate others. There is a gap in our knowledge about how this process happens. While other forms of architectural representation have a rich history of critical enquiry to draw upon, my research indicates that software engineering may offer a similar foundation to the practice of parametric modelling. But doing so requires shifting our focus beyond toolmaking, beyond our infatuation with what parametric models do, and towards what is, for lack of a better term, *parametric modelling*.

9 Conclusion

In many ways the conclusion to this thesis is simple: software engineers creating computer programs and architects designing with parametric models share similar challenges, which can often be addressed with similar research methods and similar design practices.

But this simplicity can be hard to discern. Fifty years ago when Timothy Johnson dragged a pen across the flickering screen of Sketchpad, it looked like he was drawing. Today many would say Johnson was *toolmaking*, almost as if making a tee-square is somehow a precedent for weaving a parametric model from a network of explicit functions. However, unlike the tee-square, or any other prior form of design representation, parametric models merge making and using to the point of indistinguishability. This presents unfamiliar challenges to designers; challenges that have been causing setbacks on numerous architecture projects. These challenges resemble challenges faced in software engineering. My research suggests that such an association offers a proven pathway both for conducting parametric modelling research and for improving the practice of parametric modelling with aspects of the software engineering body of knowledge. Admittedly there is something counterintuitive to the notion that programmers can teach architects about contemporary design representation but, while it can be hard to discern, in some respects the contemporary practice of architecture has more in common with the software engineers of Silicon Valley than the sketchpads used by previous generations of architects.

10 Bibliography

10.1 Published During Study

First Author Papers

Davis, Daniel, and Brady Peters. 2013. "Design Ecosystems: Customising the Design Environment with plugins." *Architectural Design* (forthcoming).

Davis, Daniel, Jane Burry, and Mark Burry. 2012. "Yeti: Designing Geometric Tools with Interactive Programming." In *Meaning, Matter, Making: Proceedings of the 7th International Workshop on the Design and Semantics of Form and Movement*, edited by Lin-Lin Chen, Tom Djajadiningrat, Loe Feijs, Simon Fraser, Steven Kyffin, and Dagmar Steffen, 196–202. Wellington, New Zealand: Victoria University of Wellington.

———. 2011a. "Untangling Parametric Schemata: Enhancing Collaboration through Modular Programming." In *Designing Together: Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures*, edited by Pierre Leclercq, Ann Heylighen, and Geneviève Martin, 55-68. Liège: Les Éditions de l'Université de Liège. Selected as the best paper of CAAD Futures 2011.

———. 2011b. "Understanding Visual Scripts: Improving collaboration through modular programming." *International Journal of Architectural Computing* 9 (4): 361-376.

———. 2011c. "The Flexibility of Logic Programming." In *Circuit Bending, Breaking and Mending: Proceedings of the 16th International Conference on Computer Aided Architectural Design Research in Asia*, edited by Christiane Herr, Ning Gu, Stanislav Roudavski, and Marc Schnabel, 29–38. Newcastle, Australia: The University of Newcastle.

Davis, Daniel, Flora Salim, and Burry Jane. 2011. "Designing Responsive Architecture: Mediating Analogue and Digital Modelling in Studio." In *Circuit Bending, Breaking and Mending: Proceedings of the 16th International Conference on Computer Aided Architectural Design Research in Asia*, edited by Christiane Herr, Ning Gu, Stanislav Roudavski, and Marc Schnabel, 155–164. Newcastle, Australia: The University of Newcastle.

Non-first Author Papers

Bohnenberger, Sascha, Chin Koi Khoo, Daniel Davis, Mette Ramsgard Thomsen, Ayelet Karmon, and Mark Burry. 2012. "Sensing Material Systems – Novel Design Strategies." *International Journal of Architectural Computing* 10 (3): 361–376.

Burry, Jane, Mark Burry, Martin Tamke, Mette Thomsen, Phil Ayres, Alexander Peña de Leon, Daniel Davis, Anders Deleuran, Stig Nielsen, Jacob Riiber. 2012. "Process through practice: synthesizing a novel design and production ecology through Dermoid." In *Synthetic Digital Ecologies: Proceedings of the 32nd Annual Conference of the Association for Computer Aided Design in Architecture*, edited by Mark Cabrinha, Jason Johnson, and Kyle Steinfeld, 127–138. San Francisco: California College of the Arts.

Burry, Jane, Daniel Davis, Brady Peters, Phil Ayres, John Klein, Alexander Peña de Leon, and Mark Burry. 2011. "Modelling Hyperboloid Sound Scattering: The challenge of simulating, fabricating and measuring." In *Computational Design Modeling: Proceedings of the Design Modeling Symposium Berlin 2011*, edited by Christoph Gengnagel, Axel Kilian, Norbert Palz, and Fabian Scheurer, 89-96. Berlin: Springer-Verlag.

Selected Blogposts (Cited in Thesis)

Davis, Daniel. 2011a. "The MacLeamy Curve." *Digital Morphogenesis*. Published 15 October. http://www.nzarchitecture.com/blog/index.php/2011/10/15/macleamy/.

———. 2011b. "Datamining Grasshopper." *Digital Morphogenesis*. Published 20 September. http://www.nzarchitecture.com/blog/index.php/2011/09/20/datamining-grasshopper/.

———. 2010. "Patrik Schumacher – Parametricism." *Digital Morphogenesis*. Published 25 September. http://www.nzarchitecture.com/blog/index.php/2010/09/25/patrik-schumacher-parametricism/.

10.2 Works Cited

ACM (Association for Computing Machinery). 2000. "A Summary of the ACM Position on Software Engineering as a Licensed Engineering Profession." Unpublished report, 17 July.

AIA (The American Institute of Architects). 2007. "Integrated Project Delivery : A Guide." Accessed 21 November 2012. http://info.aia.org/SiteObjects/files/IPD_Guide_2007.pdf.

Aberdeen Group. 2007. "The Design Reuse Benchmark Report: Seizing the Opportunity to Shorten Product Development." February. Boston: Aberdeen Group.

Abran, Alain, and James Moore, eds. 2004. *Guide to the Software Engineering Body of Knowledge: Version 2004*. Los Alamitos: Institute of Electrical and Electronic Engineers Computer Society.

Aish, Robert. 2005. "From Intuition to Precision." In *23rd eCAADe Conference Proceedings*, 62–63. Lisbon: Technical University of Lisbon.

———. 2011. "Designing at t + n." *Architectural Design* 81 (6): 20–27.

Aish, Robert, Benjamin Barnes, Mehdi Sheikholeslami, and Ben Doherty. 2012. "Multi-modal Manipulation of a Geometric Model." US Patent application 13/306,730, filed 29 November 2011, and published 12 July 2012.

Aish, Robert, and Robert Woodbury. 2005. "Multi-level Interaction in Parametric Design." In *Smart Graphics: 5th International Symposium*, edited by Andreas Butz, Brian Fisher, Antonio Krüger, and Patrick Olivier, 151–162. Frauenwörth Cloister: Springer.

Appleby, Doris, and Julius VandeKopple. 1997. *Programming Languages: Paradigm and Practice*. Second edition. Massachusetts: McGraw-Hill.

Aranda, Benjamin, and Chris Lasch. 2005. *Tooling: Pamphlet Architecture 27*. New York: Princeton Architectural Press.

Asanowicz, Alexander. 1989. "Four Easy Questions." In *Education Research and Practice: 8th eCAADe Conference Proceedings*, edited by K. Agger and U. Lentz. Aarhus: School of Architecture Aarhus.

Bagert, Donald, Thomas Hilburn, Greg Hislop, Michael Lutz, and Michael Mccracken. 1999. *Guidelines for Software Engineering Education Version 1.0*. Pittsburgh: Carnegie Mellon University.

Barrie, Donald, and Boyd Paulson. 1991. *Professional Construction Management: Including C.M., Design-construct, and General Contracting*. Third edition. Hightstown: McGraw Hill.

Beck, Kent. 1999. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, et al. 2001a. "Manifesto for Agile Software Development." Published February. http://agilemanifesto.org/.

———. 2001b. "Principles behind the Agile Manifesto." Published February. http://agilemanifesto.org/principles.html.

Bentley Systems. 2008. *GenerativeComponents V8i Essentials*. Exton, PA: Bentley Systems.

Boehm, Barry. 1976. "Software Engineering." *IEEE Transactions on Computers* 25 (12): 1226-1241.

———. 1981. *Software Engineering Economics*. Upper Saddle River: Prentice Hall.

———. 1988. "A Spiral Model of Software Development and Enhancement." *Computer* 21 (5): 61-72.

Böhm, Corrado, and Giuseppe Jacopini. 1966. "Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules." *Communications of the Association for Computing Machinery* 9 (5): 366-371.

Brooks, Frederick. 1975. *The Mythical Man-month : Essays on Software Engineering*. Anniversary edition. Boston: Addison-Wesley.

———. 2010. *The Design of Design: Essays from a Computer Scientist*. Upper Saddle River: Addison-Wesley.

Brown, Tim. 2009. *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*. New York: HarperCollins.

Brüderlin, Beat. 1985. "Using Prolog for Constructing Geometric Objects Defined by Constraints." In *Proceedings of the European Conference on Computer Algebra*, edited by Bob Caviness, 448-459. Linz: Springer.

Bucci, Federico, and Marco Mulazzani. 2000. *Luigi Moretti: Works and Writings*. New York: Princeton Architectural Press.

Burry, Jane. 2007. "Mindful Spaces: Computational Geometry and the Conceptual Spaces in which Designers Operate." *International Journal of Architectural Computing* 5 (4): 611-624.

Burry, Jane, and Mark Burry. 2006. "Sharing Hidden Power: Communicating Latency in Digital Models." In *Communicating Space(s): 24th eCAADe Conference Proceedings*. Volos: University of Thessaly.

Burry, Mark. 1996. "Parametric Design and the Sagrada Família." *Architectural Research Quarterly*, no. 1 : 70-80.

———. 2011. *Scripting Cultures*. Chichester: Wiley.

Card, Stuart, George Robertson, and Jock Mackinlay. 1991. "The Information Visualizer, an Information Workspace." In *Reaching Through Technology: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 181–188. New Orleans: Association for Computing Machinery.

Celani, Gabriela, and Carlos Vaz. 2012. "CAD Scripting And Visual Programming Languages For Implementing Computational Design Concepts: A Comparison From A Pedagogical Point Of View." *International Journal of Architectural Computing* 10 (1): 121–138.

Christensen, Henrik. 2010. *Flexible, Reliable Software: Using Patterns and Agile Development*. Boca Raton: Chapman & Hall/CRC.

CiteSeer. 2012. "Most Cited Computer Science Articles." Published 20 May. http://citeseerx.ist.psu.edu/stats/articles.

Converso, Stefano, and Fabrizio Bonatti. 2006. "Parametric Model for Architectural Design." In *Game Set and Match: On Computer Games, Advanced Geometries, and Digital Technologies*, edited by Kas Oosterhuis and Lukas Feireiss, 242–247. Rotterdam: Episode Publishers.

Creative Commons. 2007. "Attribution-ShareAlike 3.0 Unported License Deed." Accessed 19 November 2012. http://creativecommons.org/licenses/by-sa/3.0/.

Creswell, John, and Vicki Clark. 2007. *Designing and Conducting: Mixed Methods Research*. Thousand Oaks: Sage.

Cross, Nigel. 2006. *Designerly Ways of Knowing*. London: Springer-Verlag.

———. 2011. *Design Thinking: Understanding How Designers Think and Work*. Oxford: Berg.

Dana, James. 1837. "On the Drawing of Figures of Crystals." *The American Journal of Science and Arts* 32: 30-50.

DeMarco, Tom. 1982. *Controlling Software Projects: Management, Measurement, and Estimates*. New York: Yourdon Press.

———. 2009. "Software Engineering: An Idea Whose Time has Come and Gone." *IEEE Software* 26 (4): 95-96.

Dijkstra, Edsger. 1968. "Go To Statement Considered Harmful." *Communications of the Association for Computing Machinery* 11 (3): 147–148.

———. 1970. *Notes on Structured Programming*. Second Edition. Eindhoven: Technological University of Eindhoven.

———. 1972. "The Humble Programmer." *Communications of the Association for Computing Machinery* 15 (10): 859-866.

———. 1985. "Foreword." In *Communicating Sequential Processes*, by Charles Hoare, iii. Upper Saddle River: Prentice Hall.

———. 1997. "The Tide, Not the Waves." In *Beyond Calculation: The Next Fifty Years of Computing*, edited by Peter Denning and Robert Metcalfe, 59–64. New York: Springer.

Dino, Ipek. 2012. "Creative Design Exploration by Parametric Generative Systems in Architecture." *METU Journal of Faculty of Architecture* 29 (1): 207–224.

Dorfman, Merlin, and Richard Thayer. 1996. "Issues: The Software Crisis." In *Software Engineering*, edited by Merlin Dorfman, and Richard Thayer, 1–3. Los Alamitos: IEEE Computer Society Press.

Détienne, Françoise. 2001. *Software Design: Cognitive Aspects*. Translated and edited by Frank Bott. London: Springer.

Díaz-Herrera, Jorge, and Thomas Hilburn, eds. 2004. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. Engineering*. Los Alamitos: Institute of Electrical and Electronic Engineers Computer Society.

Earnshaw, Samuel. 1839. "On the Nature of the Molecular Forces which Regulate the Constitution of the Luminiferous Ether." *Transactions of the Cambridge Philosophical Society* 7: 97-112.

Eastman, Chuck, Paul Teicholz, Rafael Sacks, and Kathleen Liston. 2011. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors*. Second edition. Upper Saddle River: Wiley.

El Emam, Khaled, Saïda Benlarbi, Nishith Goel, and Shesh Rai. 2001. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics." *IEEE Transactions on Software Engineering* 27 (7): 630-650.

Erhan, Halil, Robert Woodbury, and Nahal Salmasi. 2009. "Visual Sensitivity Analysis of Parametric Design Models: Improving Agility in Design." In *Joining Languages, Cultures and Visions: Proceedings of the 13th International Conference on Computer Aided Architectural Design Futures*, edited by Temy Tidafi and Tomás Dorta, 815–829. Montreal: Les Presses de l'Université de Montréal.

Evans, Clark. 2011. "The Official YAML Web Site." Last modified 20 November. http://www.yaml.org/.

Evans, David, and Paul Gruba. 2002. *How to Write a Better Thesis*. Second edition. Melbourne: Melbourne University Press.

Eveleens, Laurenz, and Chris Verhoef. 2010. "The Rise and Fall of the Chaos Report Figures." *IEEE Software* 27 (1): 30-36.

Fischer, Thomas. 2008. "Designing (tools (for designing (tools for ...)))." PhD dissertation, Royal Melbourne Institute of Technology.

Fudos, Ioannis. 1995. "Constraint Solving for Computer Aided Design." PhD dissertation, Purdue University.

Gage, Mark. 2011a. "Project Mayhem." *Fulcrum*, 8 June.

———. 2011b. "Software Monocultures." In *Composites, Surfaces, and Software: High Performance Architecture*, edited by Greg Lynn and Mark Gage, 107–120. New Haven: Yale School of Architecture.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns*. Boston: Addison-Wesley.

Gerber, David. 2007. "Parametric Practices: Models for Design Exploration in Architecture." PhD dissertation, Harvard University.

Glegg, Gordon. 1969. *The Design of Design*. Cambridge: Cambridge University Press.

Gonzalez, Camacho, M. Williams, and E. Aitchison. 1984. "Evaluation of the Effectiveness of Prolog for a CAD Application." *IEEE Computer Graphics and Applications* 4 (3): 67-75.

Green, Thomas, and Marian Petre. 1996. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." *Journal of Visual Languages & Computing* 7 (2): 131-174.

HOK. 2012. "HOK History + Lore." Accessed 21 November 2012. http://www.hok.com/lore/.

Hatherley, Owen. 2010. "Zaha Hadid Architects and the Neoliberal Avant-Garde." *MUTE*. First published 26 October. http://www.metamute.org/editorial/articles/zaha-hadid-architects-and-neoliberal-avant-garde. Subsequently published in 2011, *MUTE* 3 (1): 39-53. This print edition does not contain the paragraph cited. Hatherley was unaware of this change (per. comm. 15 August 2011). The online edition was later updated to match the print edition, although the paragraph remains in translations of the article, such as: 2011. "Zaha Hadid y la Vanguadia Neoliberal." *Tintank*. Published 15 January. http://www.tintank.es/?p=736

Helm, Richard, and Kim Marriott. 1986. "Declarative Graphics." In *Proceedings of the Third International Conference on Logic Programming*, 513-527. London: Springer.

Henderson-Sellers, Brian, and David Tegarden. 1994. "The Theoretical Extension of Two Versions of Cyclomatic Complexity to Multiple Entry/exit Modules." *Software Quality Journal* 3 (4): 253–269.

Hewitt, Carl. 2009. "Middle History of Logic Programming: Resolution, Planner, Edinburgh LCF, Prolog, Simula, and the Japanese Fifth Generation Project." *arXiv preprint arXiv:0904.3036*.

Hilburn, Thomas, Iraj Hirmanpour, Soheil Khajenoori, Richard Turner, and Abir Qasem. 1999. *A Software Engineering Body of Knowledge Version 1.0*. Pittsburgh: Carnegie Mellon University.

Holzer, Dominik, Richard Hough, and Mark Burry. 2007. "Parametric Design and Structural Optimisation for Early Design Exploration." *International Journal of Architectural Computing* 5 (4): 625-644.

Hudson, Roland. 2010. “Strategies for Parametric Design in Architecture: An Application of Practice Led Research.” PhD dissertation, University of Bath.

ISO (International Organisation for Standards). 2000. *Information Technology: Software product quality – Part 1: Quality model*. ISO/IEC 9126. New York: American National Standards Institute.

Janssen, Patrick, and Kian Chen. 2011. “Visual Dataflow Modeling : A Comparison of Three Systems.” In *Designing Together: Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures*, edited by Pierre Leclercq, Ann Heylighen, and Geneviève Martin, 801–816. Liège: Les Éditions de l’Université de Liège.

Jenett, Florian. 2012. “Simplelivecoding Source.” *Github*. Last modified 13 November. https://github.com/fjenett/simplelivecoding.

Keller, Sean. 2006. “Fenland Tech: Architectural Science in Postwar Cambridge.” *Grey Room* 23 (April): 40-65.

Kemmis, Stephen, and Robin McTaggart. 1982. *The Action Research Planner*. Geelong: Deakin University.

Khabazi, Zubin. 2010. “Generative Algorithms using Grasshopper.” *Morphogenesism*. Accessed 19 November. http://www.morphogenesism.com/generative-algorithms.html.

Kilian, Axel. 2006. “Design Exploration through Bidirectional Modeling of Constraints.” PhD dissertation, Massachusetts Institute of Technology.

Kilpatrick, James. 1984. *The Writer’s Art*. Kansas City: Andrews, McMeel, and Parker.

Knuth, Donald. 1968. *Art of Computer Programming*. Upper Saddle River: Addison-Wesley.

Ko, Andrew. 2010. “Understanding Software Engineering Through Qualitative Methods.” In *Making Software: What Really Works, and Why We Believe It*, edited Andy Oram and Greg Wilson, 55–64. Sebastopol: O’Reilly.

Kolarić, Siniša, Halil Erhan, Robert Woodbury, and Bernhard Riecke. 2010. “Comprehending Parametric CAD Models: An Evaluation of Two Graphical User Interfaces.” In *Extending Boundaries: Proceedings of the 6th Nordic Conference on Human-Computer Interaction*, edited by Ann Blandford, Jan Gulliksen, Ebba Hvannberg, Marta Larusdottir, Effie Law, and Hannes Vilhjalmsson, 707–710. New York: Association for Computing Machinery.

Kowalski, Robert. 1988. “The Early Years of Logic Programming.” *Communications of the Association for Computing Machinery* 31 (1): 38-43.

Lally, Adam, and Paul Fodor. 2011. “Natural Language Processing With Prolog in the IBM Watson System.” *The Association for Logic Programming (ALP) Newsletter*. Published 31 March. http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/.

Lawson, Bryan. 2005. *How Designers Think: The Design Process Demystified*. Fourth edition. Oxford: Elsevier. First published 1980.

Leach, Neil, and Patrik Schumacher. 2012. “On Parametricism: A Dialogue Between Neil Leach and Patrik Schumacher.” Accessed 16 November 2012. http://www.patrikschumacher.com/Texts/On%20Parametricism%20-%20A%20Dialogue%20between%20Neil%20Leach%20and%20Patrik%20Schumacher.html. Originally published in *Time + Architecture* no. 2012/5: 1–8.

Leitão, António, Luís Santos, and José Lopes. 2012. “Programming Languages for Generative Design: A Comparative Study.” *International Journal of Architectural Computing* 10 (1): 139–162.

Leslie, John. 1821. *Geometrical Analysis and Geometry of Curve Lines*. Second edition. Edinburgh: W. & C. Tait.

Lewis, Clayton, and John Rieman. 1993. “Task-Centered User Interface Design: A Practical Introduction.” Accessed 19 November 2012. http://hcibib.org/tcuid/tcuid.pdf.

Lincke, Rudiger, and Welf Lowe. 2007. *Compendium of Software Quality Standards and Metrics*. Version 1.0. Published 3 April. http://www.arisa.se/compendium/.

Lincoln Laboratory. 1964. *Computer Sketchpad*. Digitised copy of original. Youtube video, posted by "bigkif," 17 November 2007, https://www.youtube.com/watch?v=USyoT_Ha_bA.

Livingston, Mike. 2002. "Watergate: The name that branded more than a building." *Washington Business Journal*, 17 June.

Loukissas, Yanni. 2009. "Keepers of the Geometry." In *Simulation and Its Discontents*, edited by Sherry Turkle, 153–70. Massachusetts: MIT Press.

MacLeamy, Patrick. 2010. "Bim-Bam-Boom! How to Build Greener, High-performance Buildings." *HOK Renew*. Accessed 21 November 2012. http://hokrenew.com/2010/02/09/bim-bam-boom-how-to-guarantee-greener-high-performance-buildings/.

Makris, Dimitrios, Ioannis Havoutis, Georges Miaoulis, and Dimitri Plemenos. 2006. "MultiCAD: MOGA A System for Conceptual Style Design of Buildings." In *Proceedings of the 9th 3IA International Conference on Computer Graphics and Artificial Intelligence*, edited by Dimitri Plemenos, 73-84.

Martin, Philippe, and Dominique Martin. 1999. "PolyFormes: Software for the Declarative Modelling of Polyhedra." *The Visual Computer* 15 (2): 55-76.

McCabe, Thomas. 1976. "A Complexity Measure." *IEEE Transactions on Software Engineering* 2 (4): 308-320.

McCartney, James. 2002. "Rethinking the Computer Music Language: Super Collider." *Computer Music Journal* 26 (4): 61–68.

McConnell, Steven. 2006. *Software Estimation: Demystifying the Black Art*. Redmond: Microsoft Press.

Menzies, Tim, and Forrest Shull. 2010. "The Quest for Convincing Evidence." In *Making Software: What Really Works, and Why We Believe It*, edited Andy Oram and Greg Wilson, 3-11. Sebastopol: O'Reilly.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. Second edition. Upper Saddle River: Prentice Hall.

Microsoft. 2005. "Testing Methodologies." *Microsoft Developers Network*. Published January. http://msdn.microsoft.com/en-us/library/ff649520.aspx.

Miller, Robert. 1968. "Response Time in Man-Computer Conversational Transactions." In *Proceedings of the AFIPS Fall Joint Computer Conference*, 267–277. Washington, DC: Thompson.

Mitchell, William. 1990. *The Logic of Architecture: Design, Computation, and Cognition*. Massachusetts: MIT Press.

Monedero, Javier. 1997. "Parametric Design. A Review and Some Experiences." In *Challenges of the Future: 15th eCAADe Conference Proceedings*, edited by Bob Martens, Helena Linzer, and Andreas Voigt. Vienna: Österreichischer Kunstund Kulturverlag.

Moretti, Luigi. 1957. "Forma Come Structtura." *Spazio* June-July. Republished in: Federico Bucci and Marco Mulazzani. 2000. *Luigi Moretti: Works and Writings*. New York: Princeton Architectural Press. Citations refer to the 2000 publication.

———. 1971. "Ricerca Matematica in Architettura e Urbanisticâ." *Moebius IV* no. 1, 30-53. Republished in: Federico Bucci and Marco Mulazzani. 2000. *Luigi Moretti: Works and Writings*. New York: Princeton Architectural Press. Citations refer to the 2000 publication.

Márkusz, Zsuzsanna. 1982. "Design in Logic." *Computer-Aided Design* 14 (6): 335-343.

Nasirova, Diliara, Halil Erhan, Andy Huang, Robert Woodbury, and Bernhard Riecke. 2011. "Change Detection in 3D Parametric Systems: Human-Centered Interfaces for Change Visualization." In *Designing Together: Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures*, edited by Pierre Leclercq, Ann Heylighen, and Geneviève Martin, 751–764. Liège: Les Éditions de l'Université de Liège.

Naur, Peter, and Brian Randell, eds. 1968. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*. Garmisch: Scientific Affairs Division, NATO.

Nielsen, Jakob. 1993. *Usability Engineering*. San Diego: Morgan Kaufmann.

———. 1994. "Guerrilla HCI: Using Discount Usability Engineering to Penetrate the Intimidation Barrier." In *Cost-justifying Usability*, edited by Randolph Bias and Deborah Meyhew: 245-272. San Diego: Morgan Kaufmann.

Otto, Frei, and Bodo Rasch. 1996. *Finding Form: Towards an Architecture of the Minimal*. Stuttgart: Axel Menges.

PTC (Parametric Technology Corporation). 2008. “Explicit Modeling: What To Do When Your 3D CAD Productivity Isn’t What You Expected.” White-paper. Needham: Parametric Technology Corporation.

Parnas, David. 1972. “On the Criteria To Be Used in Decomposing Systems into Modules.” *Communications of the Association for Computing Machinery* 15 (12): 1053-1058.

Patrikalakis, Nicholas. 1993. “Surface-to-Surface Intersections.” *Computer Graphics and Applications* 13 (1): 89–95.

Paulson, Boyd. 1976. “Designing to Reduce Construction Costs.” *Journal of the Construction Division* 102 (4): 587–592.

Payne, Andrew, and Rajaa Issa. 2009. “Grasshopper Primer.” Second edition. *LIFT Architects*. Accessed 19 November. http://www.liftarchitects.com/downloads/.

Payne, James. 2010. *Beginning Python: Using Python 2.6 and Python 3.1*. Indiana: Wiley.

Peters, Brady, and Tobias Olesen. 2010. “Integrating Sound Scattering Measurements in the Design of Complex Architectural Surfaces.” In *Future Cities: 28th eCAADe Conference Proceedings*, 481–491. Zurich: ETH Zurich.

Philipson, Graeme. 2005. “A Short History of Software.” In *Management, Labour Process and Software Development: Reality Bites*, edited by Rowena Barrett, 12-39. London: Routledge.

Phillips, W. 1974. “On the Distinction Between Sensory Storage and Short-term Visual Memory.” *Perception and Psychophysics* 16 (2): 283–290.

Piker, Daniel. 2011. “Using Kangaroo (Grasshopper Version) (DRAFT).” Accessed 19 November 2012. https://docs.google.com/document/preview?id=1X-tW7r7tfC9duICi7XyI9wmPkGQUPIm_8sj7bqMvTXs.

RTC (Revit Technology Corporation). 2000a. “Revit Technology Corporation Launches Industry’s First Parametric Building Modeler.” Press release, 5 April. Waltham.

———. 2000b. “Revit Technology Corporation - Product.” http://revit.com/cornerstone/index.html. Copy archived 10 May 2000. http://web.archive.org/web/20000510111053/http://revit.com/cornerstone/index.html.

Read, Phil, James Vandezande, and Eddy Krygiel. 2012. *Mastering Autodesk Revit Architecture 2013*. Indianapolis: Wiley.

Rittel, Horst, and Melvin Webber. 1973. “Dilemmas in a General Theory of Planning.” *Policy Sciences* 4(1973): 155–169.

Royce, Winston. 1970. “Managing the Development of Large Software Systems.” In *Proceedings of IEEE WESCON*, 328–338.

Rubin, Frank. 1987. “‘GOTO Considered Harmful’ Considered Harmful.” *Communications of the Association for Computing Machinery* 30 (3): 195–196.

Ruiter, Maurice, ed. 1988. *Advances in ComputerGraphics III*. Berlin: Springer-Verlag.

Rutten, David. 2012. “Programming, Conflicting Perspectives.” *I Eat Bugs For Breakfast*. Published 1 April. http://ieatbugsforbreakfast.wordpress.com/2012/04/01/programming-conflicting-perspectives/.

Scheurer, Fabian, and Hanno Stehling. 2011. “Lost in Parameter Space ?” *Architectural Design* 81 (4): 70-79.

Schultz, Carl, Robert Amor, and Hans Guesgen. 2009. “Unit Testing for Qualitative Spatial and Temporal Reasoning.” In *Proceedings of the Twenty-Second International Florida Artificial Intelligence Research Society Conference*, edited by Lane Chad and Hans Guesgen, 402–407. Florida: AAAI Press.

Schumacher, Patrik. 2008. “Parametricism as Style: Parametricist Manifesto.” Paper presented at *The Darkside Club*, 11th Architecture Biennale, Venice, 11 September. Digital copy of text, accessed 17 November 2012. http://www.patrikschumacher.com/Texts/Parametricism as Style.htm.

———. 2009a. “Parametricism: A New Global Style for Architecture and Urban Design.” *Architectural Design* 79 (4): 14–23.

———. 2009b. “Parametricism.” Keynote presentation at *Intensive Fields*, University of Southern California, Los Angeles. Digital video of presentation, accessed 17 November 2012. http://www.patrikschumacher.com/Videos/vid_01_intensive.html.

———. 2010. “The Parametricist Epoch: Let the Style Wars Begin.” *Architects’ Journal* 231 (16): 41-45.

Schön, Donald. 1983. *The Reflective Practitioner: How Professionals Think in Action*. London: Maurice Temple Smith.

Seibel, Peter. 2009. *Coders at Work: Reflections on the Craft of Programming*. New York: Apress.

Sharp, John. 1992. “A Brief Introduction to Data Flow.” In *Dataflow Computing: Theory and Practice*, edited by John Sharp. Norwood: Ablex.

Shelden, Dennis. 2002. “Digital Surface Representation and the Constructibility of Gehry’s Architecture.” PhD dissertation, Massachusetts Institute of Technology.

Side Effects Software. 2012. “Houdini User Guide.” Accessed Dec 20. http://www.sidefx.com/index.php?option=com_content&task=blogcategory&id=192&Itemid=346

Simons, Daniel, and Daniel Levin. 1997. “Change Blindness.” *Trends in Cognitive Sciences* 1 (7): 261–267.

Smith, Rick. 2007. “Technical Notes From Experiences and Studies in Using Parametric and BIM Architectural Software.” Published 4 March. http://www.vbtllc.com/images/VBTTechnicalNotes.pdf.

———. 2010. “About Virtual Build Technologies.” Accessed 19 November 2012. http://www.vbtllc.com/index_about.html.

Sorensen, Andrew. 2005. “Impromptu : An Interactive Programming Environment for Composition and Performance.” In *Generate and Test: Proceedings of the Australasian Computer Music Conference 2005*, 149-154. Brisbane: ACMA.

Sorensen, Andrew, and Henry Gardner. 2010. “Programming With Time: Cyber-physical Programming with Impromptu.” In *Onward! Proceedings of the Association for Computing Machinery International Conference on Object Oriented Programming Systems Languages and Applications*, 822–834. Tahoe: Association for Computing Machinery.

Stake, Robert. 2005. “Qualitative Case Studies.” In *The SAGE Handbook of Qualitative Research*, edited Norman Denzin and Yvonnas Lincoln, 443-466. Third edition. Thousand Oaks: Sage.

Sterling, Leon, and Ehud Shapiro. 1994. *The Art of Prolog: Advanced Programming Techniques*. Second edition. Massachusetts: MIT Press.

Stiles, Robert. 2006. “Aggregation Strategies.” Masters dissertation, University of Bath.

Summit, Steve. 1996. *C Programming FAQs: Frequently Asked Questions*. Reading, MA: Addison-Wesley.

Sutherland, Ivan. 1963. “Sketchpad: A Man-Machine Graphical Communication System.” PhD dissertation, Massachusetts Institute of Technology.

Sutherland, Jeff, and Ken Schwaber. 2011. “The Scrum Guide: The Definitive Guide to Scrum – The Rules of the Game.” White-paper. Scrum.org.

Swinson, Peter. 1982. “Logic Programming: A Computing Tool for the Architect of the Future.” *Computer-Aided Design* 14 (2): 97-104.

TIOBE Software. 2012. “TIOBE Programming Community Index for May 2012.” Published May. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Teresko, John. 1993. “Parametric Technology Corp.: Changing the way Products are Designed.” *Industry Week*, 20 December.

Terzidis, Kostas. 2006. *Algorithmic Architecture*. New York: Architectural Press.

The Standish Group. 1994. “The CHAOS Report 1994.” White-paper. Boston: The Standish Group.

———. 2009. “The CHAOS Report 2009.” White-paper. Boston: The Standish Group.

———. 2012. “The CHAOS Report 2012.” White-paper. Boston: The Standish Group.

Van Roy, Peter. n.d. “Book Cover.” Accessed 19 November 2012. http://www.info.ucl.ac.be/~pvr/bookcover.html.

Van Roy, Peter, and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Massachusetts: MIT Press.

Victor, Bret. 2012. “Inventing on Principle.” Presentation at *Turing Complete: Canadian University Software Engineering Conference*, Montreal, 20 January. Digital video of presentation, accessed 19 November 2012. http://vimeo.com/36579366.

Walker, John. 1983. “Information Letter #10.” Internal memo at Autodesk Inc. 25 October. Subsequently published as part of John Walker ed. 1994, *The Autodesk File: Bits of History, Words of Experience*, Fourth edition, Self published with earlier editions published in 1989 by New Riders Publishing.

Wallner, Johannes, and Helmut Pottmann. 2011. “Geometric Computing for Freeform Architecture.” *Journal of Mathematics in Industry* 1 (1): 1-11.

Wang, Ge, and Perry Cook. 2004. “On-the-fly Programming : Using Code as an Expressive Musical Instrument.” In *Proceeding of the 2004 International Conference on New Interfaces for Musical Expression*, 138–143.

Watson, Arthur, and Thomas McCabe. 1996. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Gaithersburg: National Institute of Standards and Technology.

Weisberg, David. 2008. “The Engineering Design Revolution: The People, Companies and Computer Systems that Changed Forever the Practice of Engineering.” Accessed 23 July 2011. http://www.cadhistory.net.

Weisstein, Eric. 2003. *CRC Concise Encyclopedia of Mathematics*. Second edition. Boca Raton: Chapman & Hall/CRC.

West, Dave, and Tom Grant. 2010. “Agile Development : Mainstream Adoption Has Changed Agility.” White-paper. Forrester.

Whitaker, William. 1993. “Ada: The Project, The DoD High Order Language Working Group.” *ACM SIGPLAN Notices* 28 (3): 229–331.

Wirth, Niklaus. 2008. “A Brief History of Software Engineering.” *IEEE Annals of the History of Computing* 30 (3): 32-39.

Wong, Yuk Kui, and John Sharp. 1992. “A Specification and Design Methodology Based on Data Flow Principles.” In *Dataflow computing: Theory and Practice*, edited by John Sharp, 37-79. Norwood: Ablex.

Woodbury, Robert. 1990. “Variations in Solids : A Declarative Treatment.” *Computer and Graphics* 14 (2): 173-188.

———. 2010. *Elements of Parametric Design*. Abingdon: Routledge.

Woodbury, Robert, Robert Aish, and Axel Kilian. 2007. “Some Patterns for Parametric Modeling.” In *Expanding Bodies: 27th Annual Conference of the Association for Computer Aided Design in Architecture*, edited by Brian Lilley and Philip Beesley, 222–229. Halifax, Nova Scotia: Dalhousie University.

Yessios, Chris. 2003. “Is There More to Come?” In *Architecture in the Digital Age: Design and Manufacturing*, edited by Branko Kolarevic, 259–68. New York: Spon Press.

Young, Michal, and Stuart Faulk. 2010. “Sharing What We Know About Software Engineering.” In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010*, edited by Gruia-Catalin Roman and Kevin Sullivan, 439–442. Santa Fe: Association for Computing Machinery.

van der Meulen, Meine, and Miguel Revilla. 2007. “Correlations Between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs.” In *Proceedings of the 18th IEEE International Symposium on Software Reliability*, 203–208. Sweden: IEEE Computer Society Press.

10.3 Illustration Credits